TURING 图灵程序设计丛书

Feisty Duck LONDON

Bulletproof SSL and TLS

Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications

HTTPS权威指南

在服务器和Web应用上部署SSL/TLS和PKI

【英】Ivan Ristić / 著　杨洋 李振宇 蒋锷 周辉 陈传文 / 译



- Web应用防火墙技术世界级专家实战经验总结
- 阿里巴巴一线技术高手精准演绎
- 用HTTPS加密网页，让用户数据通信更安全

中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

图灵程序设计丛书

HTTPS权威指南

在服务器和Web应用上部署SSL/TLS和PKI

【英】Ivan Ristić / 著　杨洋 李振宇 蒋锷 周辉 陈传文 / 译

Bulletproof SSL and TLS

Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications

人民邮电出版社
北　京

图书在版编目（C I P）数据

HTTPS权威指南 ： 在服务器和Web应用上部署SSL/TLS和PKI / （英）伊万·里斯蒂奇著 ； 杨洋等译. -- 北京 ： 人民邮电出版社, 2016.9
（图灵程序设计丛书）
ISBN 978-7-115-43272-8

Ⅰ. ①H… Ⅱ. ①伊… ②杨… Ⅲ. ①计算机网络一网络安全一指南 Ⅳ. ①TP393.08-62

中国版本图书馆CIP数据核字(2016)第193189号

内 容 提 要

本书是集理论、协议细节、漏洞分析、部署建议于一体的详尽 Web 应用安全指南。书中具体内容包括：密码学基础，TLS 协议，PKI 体系及其安全性，HTTP 和浏览器问题，协议漏洞；最新的攻击形式，如 BEAST、CRIME、BREACH、Lucky 13 等；详尽的部署建议；如何使用 OpenSSL 生成密钥和确认信息；如何使用 Apache httpd、IIS、Nginx 等进行安全配置。

本书适合 Web 开发人员、系统管理员和所有对 Web 应用安全感兴趣的读者。

◆ 著　　[英] Ivan Ristić
译　　杨 洋　李振宇　蒋 锷　周 辉　陈传文
责任编辑　朱 巍
执行编辑　杨 琳　赵瑞琳
责任印制　彭志环

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编 100164　电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000 1/16
印张：27.25
字数：644千字　　2016年 9 月第 1 版
印数：1 – 4 000册　　2016年 9 月北京第 1 次印刷

著作权合同登记号　图字：01-2015-7645号

定价：99.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

前　　言

各位读者即将踏上密码学的神秘旅程。我已经结束了我的旅程，完成了本书；这是一段令人惊叹的经历。尽管我从SSL诞生之初就开始使用它，但直到2004年开始撰写第一本书*Apache Security*时，才对它产生了浓厚的兴趣。在五年之后的2009年，我决定寻找一些新的事情去做，于是决定在SSL上投入更多时间。自此我便专注于此，随后写成了本书。

我重新关注SSL的主要原因是我认为可以改善它：这一项重要的技术由于工具和文档的缺乏而受到阻碍。密码学这门学科非常吸引人，对这个领域了解得越多，实际上就越不了解它。换言之，了解得越多，就会发现有更多仍不了解的部分。当我对某个复杂命题的理解达到新的境界以后，另一层复杂性就随之展现在我面前。这种情况我已经数不清遇到过多少次，而这也正是密码学的迷人之处。

我花了近两年时间撰写本书。起初我认为只要花一部分精力就可以了，并没有全身心投入，但是不行。有一天我突然意识到，这个领域的变化实在太快，原本“已经完成”的章节到后来不得不返工甚至重写。到了最后，大约六个月以前，为了跟上变化，我开始抓紧每一刻闲暇撰写本书。

我写作的初衷是为了节省各位读者的时间。最近五年，我尽我所能学习了关于SSL/TLS和PKI的所有知识，我想只有少数人能像我这样付出。如果我将学到的最重要的部分写进书中，也许其他人的学习能够事半功倍——这一想法促成了本书的问世。

本书的英文标题中包含bulletproof（刀枪不入）这个单词，但这并不意味着TLS是坚不可摧的。它真正的含义是，如果读者遵循本书的指引，就可以最大化地利用TLS，并且可以将TLS部署得非常安全，让自己毫不逊色于世界上的其他任何人。这通常并不容易，尤其是针对Web应用，但如果读者坚持本书所倡导的原则，就可以拥有比其他99.99%的服务器更好的安全性。事实上，即使只付出一点点努力，拥有的安全性也会比互联网上99%的服务器更好。

一般来说，读者可以通过两种方式阅读本书。(1) 按部就班地从第1章开始。如果时间充足，通过这种方式能更好地理解本书的内容。(2) 如果需要快速解决问题，那么请直接阅读第8章和第9章。这两章阐述了如何在获得良好性能的同时部署安全服务器，包含读者需要了解的所有细节。如果之后有需要，可以将第1章至第7章的内容作为参考，将第10章至第16章的内容作为实际操作的指导。

范围和读者

本书的存在是为了记录在实际的日常工作中需要了解的关于SSL/TLS和PKI的一切。我着眼于正确地结合理论、协议细节、漏洞和弱点信息以及部署建议，以帮助读者完成工作。

在撰写本书时，我想象有以下三个群体的代表密切注视着我，希望我回答他们提出的问题。

- 系统管理员

  系统管理员一贯时间有限而被迫处理系统上日益增加的安全问题，因此他们需要有关TLS的可靠指导，从而可以快速有效地配置他们的服务器。盲目相信从网络上找到的信息往往适得其反，因为有非常多不正确和过时的信息。

- 开发人员

  虽然SSL从诞生之初就承诺为基于TCP的协议提供透明的安全保障，但现实情况却是开发人员在确保应用安全的任务中扮演着十分重要的角色。对Web应用而言尤其如此，因为Web应用一边围绕着SSL和TLS构建安全，一边整合着各种会破坏安全性的特性。理论上"仅仅启用加密"即可，而实践中不仅需要启用加密，同时也需要关注许多破坏安全性的问题，不论它们是大是小。在本书中，我尽力不遗漏这类读者可能会遇到的每一个问题。

- 管理人员

  最后是管理人员，虽然他们不必关心实施细节，但仍然需要理解发生了什么并做出决策。安全领域变得越来越复杂，理解攻击和威胁往往就是其工作的一部分。解决问题的方法经常不止一种，最好的方法往往是依情况而定。

总之，本书内容全面覆盖了HTTP和Web应用，但几乎没有提及其他协议。这主要是因为浏览器。浏览器已经成为最流行的应用分发平台，而HTTP在浏览器上运用加密的方法非常特殊，带来了很多问题。因此本书才花了非常多的篇幅讲HTTP。

事实上，除了有关HTTP的章节，全书仍有三分之二的内容提供了可以应用到任何使用TLS协议的一般性指导。关于OpenSSL、Java和Microsoft的几章分别为相应的平台提供了协议的一般信息。

也就是说，如果各位读者正在寻找Web服务器以外的其他产品的配置示例，那么将无法在本书中找到。因为Web服务器主要由几大平台占据，其他类型的应用则有大量的产品。只提供Web服务器相关的最新指导已经是相当大的挑战了，毕竟Web服务器一直在不断发展。我无法在更大的范围跟进，因此有意将其他配置示例发布在网上，并希望能够在社区中起到抛砖引玉的作用，从而保持这些指导及时更新。

本书内容

本书主要内容共16章[①]，分为若干部分。这些部分彼此依托，构成一幅完整的蓝图，始于理

① 第17章为全书总结。——编者注

论介绍，最后给出实践建议。

第一部分包括第1~3章，是全书的基础，讨论了密码学、SSL、TLS和PKI。

- 第1章 SSL、TLS和密码学
  这一章首先介绍了SSL和TLS，讨论这些安全协议从哪里融入互联网基础设施；其余部分介绍密码学，并讨论了主动网络攻击者的经典威胁模型。
- 第2章 协议
  这一章讨论TLS协议的各种细节，内容覆盖了最新的TLS 1.2，并酌情提供了早期版本的信息。这一章最后包括了从SSL 3起协议演变的概述以供参考。
- 第3章 公钥基础设施
  这一章介绍互联网PKI。PKI是当今互联网上使用的主要信任模型。这一章着重介绍其标准、组织、管理、生态系统的弱点以及未来可能的改进。

第二部分包括第4~7章，详细列举各种问题。这些问题与信任基础设施、安全协议以及实现它们的库和程序有关。

- 第4章 攻击PKI
  这一章内容涉及对信任体系的攻击，涵盖了所有主要的CA事故，详细列举了种种弱点、攻击和效果。这一章以纯粹的历史视角来回顾PKI体系的安全性，这对于理解其发展非常重要。
- 第5章 HTTP和浏览器问题
  这一章讲述HTTP和TLS之间的关系，在Web出现根本性增长后产生的问题，以及不同的网络体系之间杂乱的相互作用。
- 第6章 实现问题
  这一章内容包括随机数生成、证书验证以及其他核心TLS和PKI功能的设计和编码错误。此外，还讨论了自愿协议降级和截断攻击，涵盖心脏出血、FREAK和Logjam等备受瞩目的攻击。
- 第7章 协议攻击
  这是本书中篇幅最长的一章，涵盖了近年来发现的所有主要的协议缺陷：不安全的重新协商、BEAST、CRIME、Lucky 13、POODLE和POODLE TLS、RC4、TIME和BREACH，以及三次握手攻击。这一章还简单讨论了Bull Run项目及其对于TLS安全性的影响。

第三部分包括第8~10章，为以安全且高效的方式部署TLS提供了综合指导。

- 第8章 部署
  这一章是整本书的地图，提供了如何一步一步地部署安全且工作良好的TLS服务器和Web应用的操作指南。
- 第9章 性能优化
  这一章着眼于TLS的速度，为那些希望从服务器中榨取每一点速度的读者详述了各种提升性能的技术。
- 第10章 HTTP严格传输安全、内容安全策略和钉扎
  这一章涉及加强Web应用的一些高级主题，比如HTTP严格传输安全和内容安全策略。这

一章也涉及钉扎，它是减少由当前PKI模型带来的庞大攻击面的有效方法。

第四部分是最后一部分，由第11~16章组成，为在主流部署平台和Web服务器上使用和配置TLS以及如何使用OpenSSL探测服务器配置提供可行的指导。

- 第11章　OpenSSL

  这一章描述了OpenSSL最常用的功能，主要介绍其安装、配置、私钥和证书管理。最后的11.4节给出如何构建和管理一个私有证书颁发机构的操作指南。
- 第12章　使用OpenSSL进行测试

  这一章继续介绍OpenSSL，解释如何使用它的命令行工具测试服务器配置。尽管使用自动化工具测试经常更加简单，但当读者想确定当前具体的情况时，仍然可以使用OpenSSL工具。
- 第13章　配置Apache

  这一章讨论流行的Web服务器Apache httpd的TLS配置。从这一章开始，将有一系列章节提供实践指导，与前面章节的理论相配合。每一章都专注于一个主要的技术层面。
- 第14章　配置Java和Tomcat

  这一章涉及Java（第7版和第8版）和Tomcat服务器。除了配置信息，这章也包含Web应用安全的指导。
- 第15章　配置Microsoft Windows和IIS

  这一章讨论在Microsoft Windows平台和Internet Information Server上部署TLS的问题，也为在ASP.NET下运行的Web应用上使用TLS提供指导。
- 第16章　配置Nginx

  这一章讨论Nginx服务器，除了阐述最新稳定版的特性，也会对开发分支中的一些改进一探究竟。

SSL与TLS

非常不幸，本身相同的协议却有两个名称。根据我的经验，大多数人很熟悉在传输层加密的场景中使用SSL这个名称。有一些人，通常是那些在协议上花费更多时间的人，会使用或尝试使用更适合特定场景的正确名称。虽然也许并不能成功做到，但我仍尝试这样做。有时这显得很烦琐，但我通过下面的步骤做到了这一点：(1) 尽量不单独提及某一个名称；(2) 如果某条指导建议适用于所有版本，那么将两个名称都列出来；(3) 其他情况都使用TLS。各位读者可能不会注意到这些，那也没关系。

SSL Labs

SSL Labs（www.ssllabs.com）是我从2009年开始的一个研究项目，关注SSL/TLS和PKI的实用领域。我于2010年携带此项目加入Qualys公司。开始，我的主要职责并不在此，但是到了2014年，我将全部精力放在了SSL Labs上。

这个项目的出现主要是由于我了解到，缺乏优秀的文档和工具是导致TLS服务器配置普遍很糟的主要原因（默认配置很差是另一个重要原因）。我想，如果不能将问题可视化，我们就不能开始解决问题。多年以后，SSL Labs扩展成为以下四个主要项目。

- 服务器测试

  SSL Labs的主要特性是服务器测试，这可以帮助网站访问者审阅任何公开的Web服务器的配置。该测试包括一些其他网站都没有的重要检查，并提供服务器配置的综合视图。等级系统也很容易理解，可以帮助那些并非安全专家的人区分问题的大小。测试中最有用的部分之一是模拟握手，它可以预测在大约40种最广泛使用的程序和设备上的协商协议和加密套件。该功能可以有效地猜测TLS配置。在我看来，该功能是不可缺少的。
- 客户端测试

  客户端测试是一个较新的补充，并非众所周知，但是它仍然非常有用，主要目的是帮助我们了解客户端在众多设备上的能力。这个测试获得的结果将用于驱动服务器测试中的握手模拟器。
- 最佳实践

  “SSL/TLS部署最佳实践”是一个简洁而合理的综合指南。它为TLS服务器的配置提供了明确指导。这篇文档非常简短（英文版约14页），能够让读者在短时间内吸收，并可以作为服务器测试伴侣。
- SSL Pulse

  最后，SSL Pulse的设计旨在监控整个SSL体系，并通知我们整个体系的运转情况。它从2012年开始运作，专注于一个由启用了TLS的网站组成的核心群体，而这些网站是从Alexa排行前一百万的网站中选出来的。从那时起，SSL Pulse一直为这个关键体系统计每月快照。

还有其他一些小的项目，读者可以从SSL Labs的网站上找到关于它们的更多信息。

在线资源

本书没有在线伴侣（虽然你可以认为SSL Labs就是），但是它确实有一个包含全书中引用文档的在线存储库：github.com/ivanr/bulletproof-tls。我会及时扩展这个存储库，使它容纳其他有用的内容作为本书的补充。

如果希望得到事件和消息通知，请在Twitter上关注@ivanristic。TLS是我在这些日子里的一切工作，我努力在这里通报每一件有关联的事件。这里几乎没有垃圾信息。此外，如果本书的内容有改进，我也会在Twitter上提醒大家。

我的博客地址是blog.ivanristic.com。我会在这里对SSL体系的重要新闻和发现作出回应，宣布SSL Labs的改进，公布我的研究。

如果你购买的是本书的英文电子版，那么可以随时重新登录你在Feisty Duck网站上的账户，下载最新的版本。一次购买包括对同一版次更新的无限次访问权限。你还可以选择在本书增加新内容或者发生某些非常重要的事情（比如，发现某个新的协议缺陷）时，收到临时电子邮件通知。

反馈

我很幸运，只要我想更新本书，就可以随时更新。这并不是一个巧合，我是专门这样做的。如果我今天修改了，经过每天的自动化构建，明天大家就能看到。更新纸质的书相对麻烦，不过有了按需印刷，我们可以在需要时发布修订版。

因此，与许多其他的书永远不会再版不同，读者的反馈可以影响本书。如果读者发现一处错误，那么几天之内这个错误就会得到修正。当然，对于小的改进也是一样，比如语法修改或者说明解释。如果某个平台在某些地方发生了变化，或者有新的发展，我也会跟踪到。我的目的是，只要读者有兴趣，我就会保持本书与时俱进。

请通过ivanr@webkreator.com写信给我。

关于作者

在这部分内容中，我会以第三人称介绍自己。这就是我的“官方”自传。

> Ivan Ristić是一位安全研究员、工程师、作者。他对于Web应用防火墙领域的发展，开源Web应用防火墙ModSecurity的开发，以及在SSL Labs网站上对SSL/TLS和PKI的研究、工具和指南的发表，都作出了很大的贡献，因此享誉世界。
>
> 他写过*Apache Security*、*ModSecurity Handbook*和《HTTPS权威指南》三本书，并将其发表到Feisty Duck上。Feisty Duck是他持续写作和发表的平台。Ivan是安全社区的活跃参与者，经常在Black Hat、RSA、OWASP AppSec等各种安全会议上进行演讲。他之前就职于Qualys公司，担任应用安全研究主任。

我也应该提一下《OpenSSL攻略》，这是一本免费电子书，其中包含了本书第11章和第12章的内容，以及文档“SSL/TLS部署最佳实践”的内容。[①]

致谢

虽然这是我写的一本书，但我并不是唯一的作者。本书内容建立在分散于各种书籍、标准文档、研究论文、会议演讲、博客文章，甚至Twitter上有关加密和计算机安全的大量信息之上，多得令人难以置信。是上千人的工作成就了本书。

多年来，我有幸与许多在计算机安全领域的同仁共事，他们扩展了我在这方面的知识。他们中的很多人都帮助我审阅过部分手稿。我非常感谢他们的帮忙。本书的核心章节由许多高手帮我审阅，其中不乏那些设计标准或推翻标准的大师，也有我提到的程序开发者，这令我轻松了许多。

Kenny Paterson对第7章，也就是全书最长也是最复杂的一部分，进行了仔细的审阅，这使我受益匪浅。我认为他对待我就像对待他的学生一样，我的作品也因此有了很大的改进。我花了整

① 这本小册子的中文版可在图灵社区本书主页（http://www.ituring.com.cn/book/1734）免费下载。——编者注

整一周时间才按照Kenny的意见完成了内容修改。

Benne de Weger审阅了密码学和攻击PKI的相关章节。Nasko Oskov审阅了关于Microsoft协议实现的核心章节。来自Symantec的Rick Andrews和他的同事们帮忙审阅了第4章和第5章，此外Adam Langley也提供了帮助。Marc Stevens曾经针对PKI攻击，尤其是MD5和SHA1的选择前缀攻击写信给我。Nadhem AlFardan、Thai Duong和Juliano Rizzo审阅了第7章，并且回答了我关于他们工作的提问，非常有用。Ilya Grigorik对第9章的审查非常仔细，他的建议也非常有用。Jakob Schlyter对DANE有一种特别的关注，他审阅了关于进阶话题的章节（HSTS和CSP）。Rich Bowen和Jeff Trawick审阅了第13章，Jeff还修正了Apche中有关TLS的一些问题，这使我更加努力地跟上变化。Oracle的Xuelei Fan和Erik Costlow审阅了第14章，此外还有Mark Thomas、William Sargent和Jim Manico。Andrei Popov和Ryan Hurst审阅了第15章。Maxim Dounin总是很快地回答我对Nginx的问题，并且审阅了第16章。

Vincent Bernat的microbenchmarking工具在我写关于性能的第9章时起了很大作用。

Eric Lawrence发给我无数的注释和问题，我从没想到过我会遇到这样仔细的审阅人。Eric是每位作者所梦想的那种评论家，我非常感谢他严谨的态度。

同样，十分感谢我的读者，是你们给了我非常好的意见反馈：Pascal Cuoq、Joost van Dijk、Daniël van Eeden、Stephen N. Henson博士、Brian Howson、Rainer Jung、Brian King、Hendrik Klinge、Olivier Levillain、Colm MacCárthaigh、Dave Novick、Pascal Messerli和Christian Sage。感谢你们！

我要特别感谢文字编辑Melinda Rankin，你总是能快速响应我的编辑要求，并将其并入基于DocBook的工作流。我还要感谢Qualys公司，感谢你们支持我的写作，支持SSL Labs的工作。

目　　录

第1章 SSL、TLS和密码学

我们生活在一个互联网时代。在20世纪的最后十年，互联网已经十分普及，并且永久性地改变了我们的生活方式。今天，我们依靠手机和计算机进行通信、购买商品、支付账单、旅行、工作，等等。很多人的口袋里总是装着处于开机状态的设备；我们并不只是连接到互联网，其实就是互联网的一部分。目前手机的数量已经超过了人口的数量。智能手机的数量已经达到数十亿，并且仍保持着快速增长。与此同时，诸多计划正酝酿将各种设备连接到同一网络。显然，这一切才刚刚开始。

所有连接到互联网的设备都有一个共同点，它们依赖*安全套接字层*（secure socket layer，SSL）和*传输层安全*（transport layer security，TLS）协议保护传输的信息。

1.1 传输层安全

人们最初设计互联网时，很少考虑到安全。这样的结果是，核心通信协议本质上是不安全的，只能依靠所有参与方的诚信行为。互联网在早期由少数节点（大部分是大学）构成，那时这也许行得通；但现在所有人都可以连接到互联网，这种方式便土崩瓦解。

SSL和TLS都是加密协议，旨在基于不安全的基础设施提供安全通信。这意味着，如果正确部署这些协议，你就可以对互联网上的任意一个服务打开通信信道，并且可以确信你会与正确的服务器通信，安全地交换信息（你的数据不会被他人截取，而且在接收时会保持原样）。这些协议保护着通信链路即*传输层*，这也是TLS名称的由来。

安全不是TLS的唯一目标。TLS实际上有以下四个主要目标（按优先顺序排列）。

- *加密安全*
  这是主要问题：为任意愿意交换信息的双方启用安全通信。
- *互操作性*
  独立的编程人员应该能够使用通用的加密参数开发程序和库，使它们可以相互通信。
- *可扩展性*
  你很快就会看到，TLS是一种能高效开发和部署加密协议的框架。其重要目标是独立于实际使用的加密基元（例如密码和散列函数），从而不需要创建新的协议，就允许从一个基元迁移到另一个。

- ❑ *效率*

  最终的目标是在实现上述所有目标的基础上保持性能成本在可接受的范围内。这需要尽量减少昂贵的加密操作的执行次数，并提供一个会话缓存方案，以避免这些加密操作在随后的连接中被执行。

1.2　网络层

互联网的核心是建立在IP（internet protocol）和TCP（transmission control protocol）协议之上的，这些协议用于将数据分割成小数据包进行传输。这些数据包在全世界范围内历经数千里的传输，在此期间需要跨越许多国家的许多计算机系统（称为*跃点*，hop）。由于核心协议本身不提供任何安全保障，任何有权访问通信链路的人都可以获得所有数据，并且可以在不被察觉的情况下改变这些数据。

IP和TCP不是唯一易受攻击的协议，还有一系列其他路由协议用于协助发现网络上的其他计算机。DNS和BGP就是这样的两个协议。它们同样是不安全的，可以被他人通过各种方式劫持。如果出现这种情况，发往一台计算机的连接可能由攻击者响应。

如果部署了加密，攻击者也许有能力得到加密数据的访问权限，但是不能解密数据或者篡改数据。为了避免伪装攻击，SSL和TLS依赖另外一项被称为*公钥基础设施*（public key infrastructure，PKI）的重要技术，确保将流量发送到正确的接收端。

为了理解SSL和TLS的运作，我们需要从描述网络通信的理论模型入手，即*开放系统互联*（open systems interconnection，OSI）模型，参见表1-1。简单来说，所有功能都被映射到七个层上。最底层是最接近物理通信链路的层，后面的层依次建立在其他层之上，提供更高级别的抽象。最顶层就是应用层，携带着应用数据。

> **注意**
>
> *现实中的协议并非总能与OSI模型完全对应。比如SPDY和HTTP/2因为要对连接进行管理，所以被归入会话层协议，但它们却在数据加密以后生效。第五层及更高层的划分经常是模糊的。*

表1-1　OSI模型层

层　号	OSI层	描　述	协议示例
7	应用层	应用数据	HTTP、SMTP、IMAP
6	表示层	数据表示、转换和加密	SSL / TLS
5	会话层	多连接管理	–
4	传输层	包或流的可靠传输	TCP、UDP
3	网络层	网络节点间的路由与数据分发	IP、IPSec
2	数据链路层	可靠的本地数据连接（LAN）	以太网
1	物理层	直接物理数据连接（电缆）	CAT5

以这种方式安排通信可以清晰地划分概念：高层的协议不必担心在底层实现的功能。进一步说，不同层次的协议可以加入通信或者从通信中删除，一种底层协议可以服务于多种上层协议。

SSL和TLS是这一原则如何在实践中运用的一个重要示例。它用于TCP协议之上，上层协议（如HTTP）之下。当不需要加密时，可以将TLS从模型中去掉，这并不会对上层协议产生影响（它们将直接与TCP协同工作）。当需要加密时，就可以利用TLS加密HTTP，以及其他TCP协议（比如SMTP、IMAP等）。

1.3 协议历史

SSL协议由Netscape公司开发，历史可以追溯到Netscape Navigator浏览器统治互联网的时代[①]。协议的第一个版本从未发布过，第二版则于1994年11月发布。第一次部署是在Netscape Navigator 1.1浏览器上，发行于1995年3月。

SSL 2的开发基本上没有与Netscape以外的安全专家进行过商讨，所以有严重的弱点，被认为是失败的协议，最终退出了历史的舞台。这次失败使Netscape专注于SSL 3，并于1995年年底发布。虽然名称与早先的协议版本相同，但SSL 3是完全重新设计的协议。该设计一直沿用到今天。

1996年5月，TLS工作组[②]成立，开始将SSL从Netscape迁移至IETF。由于Microsoft和Netscape当时正在为Web的统治权争得不可开交，整个迁移过程进行得非常缓慢、艰难。最终，TLS 1.0于1999年1月问世，见RFC 2246。尽管与SSL 3相比，版本修改并不大，但是为了取悦Microsoft，协议还是进行了更名[③]。

直到2006年4月，下一个版本TLS 1.1才问世，仅仅修复了一些关键的安全问题。然而，协议的重要更改是作为TLS扩展于2003年6月发布的，并被集成到了协议中，这比大家的预期早了好几年。

2008年8月，TLS 1.2发布。该版本添加了对已验证加密的支持，并且基本上删除了协议说明中所有硬编码的安全基元，使协议完全弹性化。

协议的下一个版本正在开发过程中。该版本预计会成为一个主要版本，其目标是简化设计，除去安全性较弱或者不太需要的功能，并且提升性能。各位读者可以关注TLS工作组邮件列表的讨论[④]。

---

① 若想了解更详细的SSL协议早期历史，建议阅读Eric Rescorla的*SSL and TLS: Designing and Building Secure Systems*（Addison-Wesley出版社于2001年出版），第47~51页。

② TLS工作组，https://datatracker.ietf.org/wg/tls/documents/（IETF，检索于2014年6月23日）。

③ Security Standards and Name Changes in the Browser Wars，http://tim.dierks.org/2014/05/security-standards-and-name-changes-in.html（Tim Dierks，2014年5月23日）。

④ TLS working group mailing list archives，http://www.ietf.org/mail-archive/web/tls/current/（IETF，检索于2014年7月19日）。

1.4　密码学

密码学是一门通信安全的科学，同时也是一门艺术。虽然我们总是将密码学与现代联系在一起，但在上千年以前，人们事实上就已经开始利用它的力量了。考古发现，加密工具密码棒[①]首次被提及是在公元前7世纪。我们今天所知的密码学诞生于20世纪，用于军事领域；而它现在已经成为了我们日常生活的一部分。

部署正确的密码能解决安全的三个核心需求：保持秘密（机密性）、验证身份（真实性），以及保证传输安全（完整性）。

本章剩余部分将对一个加密环境的基本构造进行讨论，展示安全性从何而来。同样，还会讨论加密体系通常是如何受到攻击的。密码学是一个非常多样化的领域，并且有非常深厚的数学基础。我会将视角保持在很宽泛的层面上，给大家介绍一些基础知识，使大家能够看懂后面的讨论。如果主题需要，我会在本书的其他部分更详细地介绍密码学的相关内容。

> **注意**
> 如果想花更多时间学习密码学，你可以找到很多文献。我最喜欢的一本书是《深入浅出密码学》（*Understanding Cryptography*，作者是Christof Paar和Jan Pelzl，2010年由Springer出版）。

1.4.1　构建基块

在最底层，使用密码加密依赖于各种加密基元（cryptographic primitive）。每种基元都着眼于某个特定功能而设计。比如，我们会使用某个基元加密，使用另外一个基元进行完整性检查。单个基元本身的作用并不大，但是我们可以将它们组合成方案（scheme）和协议（protocol），从而提供可靠的安全性。

Alice和Bob是谁？

讨论密码学时，我们为了方便起见，通常会使用Alice和Bob这两个名字[②]。他们可以使枯燥的密码学命题变得更加有趣一些。大家公认，Ron Rivest在1977年介绍RSA密码系统的论文中，首次使用了这两个名字[③]。此后，又有其他一些名字进入了密码学文化。在这一章中，我将一位具备窃听能力的攻击者命名为Eve，并将另一位能够妨碍网络流量的主动攻击者命名为Mallory。

① Scytale，https://en.wikipedia.org/wiki/Scytale（维基百科，检索于2014年6月5日）。
② Alice and Bob，https://en.wikipedia.org/wiki/Alice_and_Bob（维基百科，检索于2014年6月5日）。
③ Security’s inseparable couple，http://www.networkworld.com/article/2318241/lan-wan/security-s-inseparable-couple.html（Network World，2005）。

1. 对称加密

对称加密（symmetric encryption）又称私钥加密（private-key cryptography），是一种混淆算法，能够让数据在非安全信道上进行安全通信。为了保证通信安全，Alice和Bob首先得到双方都认可的加密算法和密钥。当Alice需要向Bob发送数据时，她使用这个密钥加密数据。Bob使用相同的密钥解密。Eve能够访问信道，所以可以看到加密数据；但因为没有密钥，所以看不到原始数据。Alice和Bob只要能保证密钥安全，就能一直安全地通信，如图1-1所示。

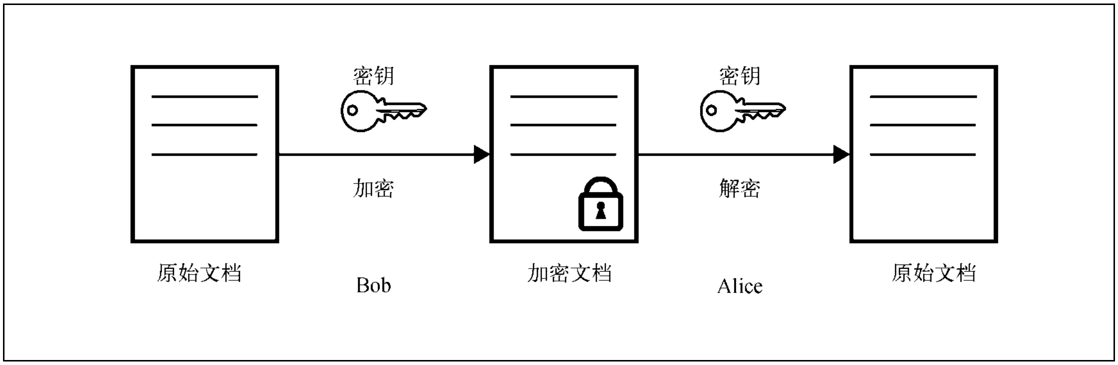


图1-1 对称加密

> **注意**
> 讨论加密时通常会使用到三个术语：**明文**（plaintext，即原始数据）、**密钥**（cipher，用于加密）和**密文**（ciphertext，即加密后的数据）。

对称加密可以追溯到上千年以前。比如，加密时将字母表中的每个字母替换成其他字母，解密时反向操作，这就是代替密码加密。在这个例子中，不存在密钥；安全性取决于保守加密方法的秘密。那就是最早的算法的例子。随着时间的流逝，我们采用了另一种方法。它是依照19世纪的一位密码破解专家Auguste Kerckhoffs的观察结果发展而来的[①]。

> 即使攻击者知晓了整个密码系统除密钥以外的所有情报，系统仍然应当能保证安全。

Kerckhoffs的原则初看起来有些奇怪，但如果继续深刻思考，就会觉得有道理，原因如下。

- 如果一种加密算法要得到广泛使用，就必须让其他人知道。当越来越多的人接触到这个算法，那么敌人得到这个算法的可能性也会增加。
- 没有密钥的简单算法非常不便于在大群体中使用；每个人都可以解密所有人的通信。
- 设计出优秀的加密算法非常困难。一种算法想要更安全，就得经过更多的曝光和审视。当需要采用一种新算法时，密码学家推荐使用保守的方法来确定算法是否安全，那就是算法需要经过许多年的破解尝试。

优秀的加密算法需要产出表面上看来随机的密文，这样攻击者就无法分析得出任何关于明文

① la cryptographie militaire，http://petitcolas.net/kerckhoffs/（Fabien Petitcolas，检索于2014年6月1日）。

的信息。比如，替换密码就不是一种好算法，因为攻击者可以确定密文中各个字母的使用频率，并将其与英语中的字母使用频率进行对比。因为某些字母比其他字母使用得更频繁，攻击者可以利用这个结果恢复明文。如果加密算法优秀，攻击者只有一种方法，那就是尝试所有可能的解码密钥，俗称*穷举密钥搜索*（exhaustive key search）。

基于这一点，我们可以说密文的安全性完全取决于密钥。如果密钥是从某个非常大的密钥空间中选取出来的，那么破解也需要遍历所有这些可能的密钥，其数量极大，几乎不可能。我们可以说这种算法在计算上是安全性的。

> **注意**
> 通常我们通过密钥长度来衡量加密强度；有一个假设是，密钥本质上是随机的，所以密钥空间才可以由密钥的位数来定义。比如，某个128位的密钥（被认为非常安全）有34×10^{37}种可能的组合。

密码可以分为两大类：序列密码和分组密码。

- *序列密码*

从概念上讲，*序列密码*（stream cipher）的操作过程与我们想象中加密的过程一致。将1字节的明文输入加密算法，就得到1字节的密文输出。在对端则进行相反的过程。整个过程持续重复，直到所有数据处理完成。

序列密码的核心是生成一串称为*密钥序列*（keystream）的无穷序列，看似杂乱无章。加密就是将密钥序列中的1字节与明文序列中的1字节进行异或操作。因为异或操作是可逆的，所以解密就是将密文序列中的1字节与密钥序列中的相同字节进行异或操作。这个过程在图1-2中描述。

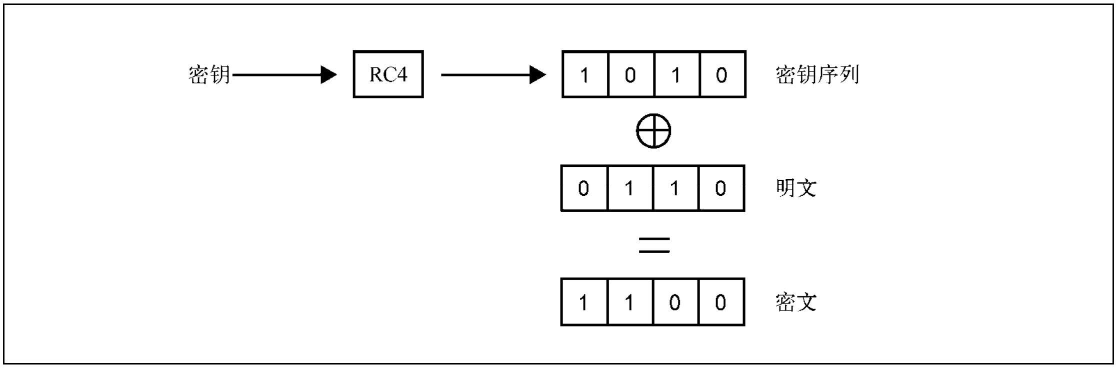


图1-2　RC4加密

只要攻击者无法预测密钥序列中对应位置的字节，就可以认为加密过程是安全的。基于这个理由，序列密码绝不能第二次使用相同的密钥，这一点非常关键。这是因为在实际使用中，攻击者知道或者可以预测特定区域的明文（请思考加密HTTP请求的情景；许多请求的请求方法、协议版本、请求头名称都是一样的）。当你知道明文，又观察到密文时，就可以解析一部分密钥序

列。如果使用了相同的密钥，那么就可以解密后续的部分密文。为了解决这个问题，序列密码都与从长期密钥中提取出来的一次性密钥一同使用。

RC4是最为人熟知的序列密码[①]。因为它很快很简单，所以一度非常流行。但是它已经不再安全。我将在7.5节中讨论它的弱点。其他现代的安全序列密码则由ECRYPT Stream Cipher Project[②]进行推进。

- 分组密码

分组密码（block cipher）每次加密一整块数据，并且现代的分组密码倾向于使用128位（16字节）大小的块。一种分组密码就是一个变换函数：接受输入并生成看似杂乱无章的输出。只要使用相同的密钥，每一个可能的输入组合都有唯一的输出。分组密码的关键特性是在输入上制造一个小变化（比如，在任意一处变换1位），从而得到大量输出变体。

分组密码本身不是非常有用，因为它们自身有一些限制。第一个问题是，只能使用它们加密长度等于加密块大小的数据。因此在实际使用分组算法时，需要一个方法处理任意长度的数据。另一个问题是，分组密码是确定的。对于相同的输入，输出也是相同的。这个特性会使许多攻击成为可能，需要解决。

实践中，人们通过称为分组密码模式（block cipher mode）的加密方案来使用分组密码。这种方案能规避这些限制，有时还可以添加身份验证。分组密码也可以作为其他加密基元的基础来使用，诸如散列函数、消息验证代码、伪随机数生成器，甚至序列密码。

世界上最流行的分组密码是高级加密标准（advanced encryption standard，AES）[③]，可以使用128位、192位和256位的加密强度。

- 填充

分组密码的挑战之一是处理数据长度小于加密块大小的数据加密。举个例子，128位的AES需要16字节的输入数据并且产出相同长度的输出。如果你的数据刚好能归入16字节的块中，那正好。但如果不足16字节，怎么办？一种方法是追加额外的数据到明文的尾部。这些额外的数据就被称为填充（padding）。

填充不能由任何随机数据构成，它必须遵循某种格式，这样接收方才可以发现填充并了解需要丢弃多少字节。在TLS中，加密块的最后1字节包含填充长度，指示填充有多少字节（不包含填充长度字节）。填充的每字节都被设置成与填充长度字节相同的值，如图1-3所示。这种方式使得接收方能够检查填充是否正确。

为了在解密后丢弃填充，接收方检查数据块的最后1字节，删除它。接着，接收方删除指定长度的字节数，同时检查它们是否都是相同的值。

---

① RC4，https://en.wikipedia.org/wiki/RC4（维基百科，检索于2014年6月1日）。

② eSTREAM: the ECRYPT Stream Cipher Project，http://www.ecrypt.eu.org/stream/（European Network of Excellence in Cryptology II，检索于2014年6月1日）。

③ Advanced Encryption Standard，https://en.wikipedia.org/wiki/Advanced_Encryption_Standard（维基百科，检索于2014年6月1日）。

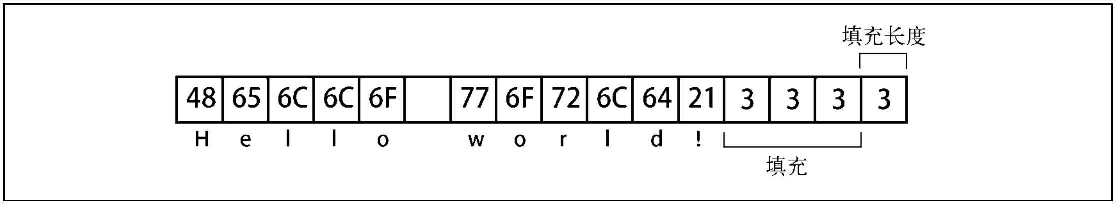


图1-3　TLS填充示例

2. 散列函数

散列函数（hash function）是将任意长度的输入转化为定长输出的算法。散列函数的结果经常被简称为散列（hash）。编程中普遍使用散列函数，但并非所有散列函数都适用于密码学。密码学散列函数有以下几个额外特性。

- 抗原像性（单向性）

  给定一个散列，计算上无法找到或者构造出生成它的消息。

- 抗第二原像性（弱抗碰撞性）

  给定一条消息和它的散列，计算上无法找到一条不同的消息具有相同的散列。

- 强抗碰撞性

  计算上无法找到两条散列相同的消息。

散列函数最常用的使用场合是以紧凑的方式表示并比较大量数据。比如，为了避免直接比较两个文件（可能很难，比方说，它们存放于世界上不同的位置），你可以比较它们的散列。散列函数经常被称为指纹、消息摘要，或者简单称为摘要。

现在使用最为广泛的散列函数是SHA1，它的输出是160位。因为SHA1已经变弱，所以建议升级为SHA256的变种。与密码不同，散列函数的强度并不与散列长度对等。因为生日悖论（概率论中的常见问题）[①]，散列函数的强度最多只是散列长度的一半。

3. 消息验证代码

散列函数可以用于验证数据完整性，但仅在数据的散列与数据本身分开传输的条件下如此。否则攻击者可以同时修改数据和散列，从而轻易地避开检测。消息验证代码（message authentication code，MAC）或者使用密钥的散列（keyed-hash）是以身份验证扩展了散列函数的密码学函数。只有拥有散列密钥，才能生成合法的MAC。

MAC通常与加密一起使用。如果没有MAC，即使Mallory无法解码密文，她也能修改传输中的数据；加密提供了机密性但无法确保完整性。如果Mallory聪明到可以修改密文，她就可以诱使Bob接受并相信伪造的消息。当MAC和密文一起发送时，（和Alice共享散列密钥的）Bob就能确认消息并未遭到篡改。

任何散列函数都能用作MAC的基础，另一个基础是基于散列的消息验证代码（hash-based

① Birthday problem，https://en.wikipedia.org/wiki/Birthday_problem（维基百科，检索于2014年6月1日）。

message authentication code，HMAC）[①]。HMAC本质就是将散列密钥和消息以一种安全的方式交织在一起。

4. 分组密码模式

分组密码模式是为了加密任意长度的数据而设计的密码学方案，是对分组密码的扩展。所有分组密码模式都支持机密性，不过有些将其与身份验证联系起来。一些模式会将分组密码转换成序列密码。

它有许多输出模式，通常以首字母缩写来引用：ECB、CBC、CFB、OFB、CTR、GCM，诸如此类（不用担心这些缩写都代表什么）。我在这里只会介绍ECB和CBC：ECB是设计一种分组加密模式的反面例子，而CBC则仍是SSL和TLS的主要模式。GCM是TLS中相对较新的模式，从1.2版本开始才能使用。它提供了机密性和完整性，是当前可用的最好模式。

- 电码本模式

电码本（electronic codebook，ECB）模式是最简单的分组密码模式。它只支持数据长度正好是块大小的整数倍的情况，如果数据长度不满足这个条件，就得事先实施填充。加密就是将数据按块大小切分，再分别加密每一块。

ECB的简单就是它的劣势。因为分组密码是确定的（输入相同，输出也相同），所以ECB也是如此。这就造成了严重的负面结果：(1) 密文中出现的模式显示出明文中对应出现的模式；(2) 攻击者可以发现信息是否重复；(3) 攻击者可以观察密文并且提交任意明文加密（在HTTP中通常是可能的，在一些其他情况下也可以），如此尝试足够的次数，就能猜出明文。这就是针对TLS的BEAST攻击的大致思路，7.2节会继续讨论。

- 加密块链接模式

加密块链接（cipher block chaining，CBC）模式是从ECB发展而来的下一步。为了解决ECB天生的确定性，CBC引入了*初始向量*（initialization vector，IV）的概念。即使输入相同，IV也可以使每次的输出都不相同，如图1-4所示。

整个过程开始于生成一个随机IV（因此不可预测），长度与加密块相等。加密前，明文第一块内容与IV进行异或操作。这一步对明文进行了掩饰，并保证密文总是不尽相同。对于下一个加密块，使用上一块的密文作为IV，以此类推。这样一来，每次加密操作都是同一个加密链条中的一部分，这也是这种模式名称的由来。至关重要的是，IV必须通过线路传送到接收端，这个信息是成功解密所必需的。

5. 非对称加密

对称加密在高速处理大量数据方面做得非常好，然而随着使用它的团体增加，产生了更多的需求，使得对称加密无法满足。

- 相同团体的成员必须共享相同的密钥。越多人加入，团体密钥出现问题的次数就越多。
- 为了更好的安全性，你可以在每两个人之间使用不同的密钥，但是这个方法不可扩展。

---

① RFC 2104: HMAC: Keyed-Hashing for Message Authentication，http://tools.ietf.org/html/rfc2104（Krawczyk等，1997年2月）。

虽然3个人只需要3个密钥，但10个人就需要45（9+8+…+1）个密钥，而1000个人需要499 500个密钥！

- 对称加密不能用于访问安全数据的无人系统。因为使用相同的密钥可以反转整个过程，这样的系统出现任何问题都会影响到存储在系统中的所有数据。

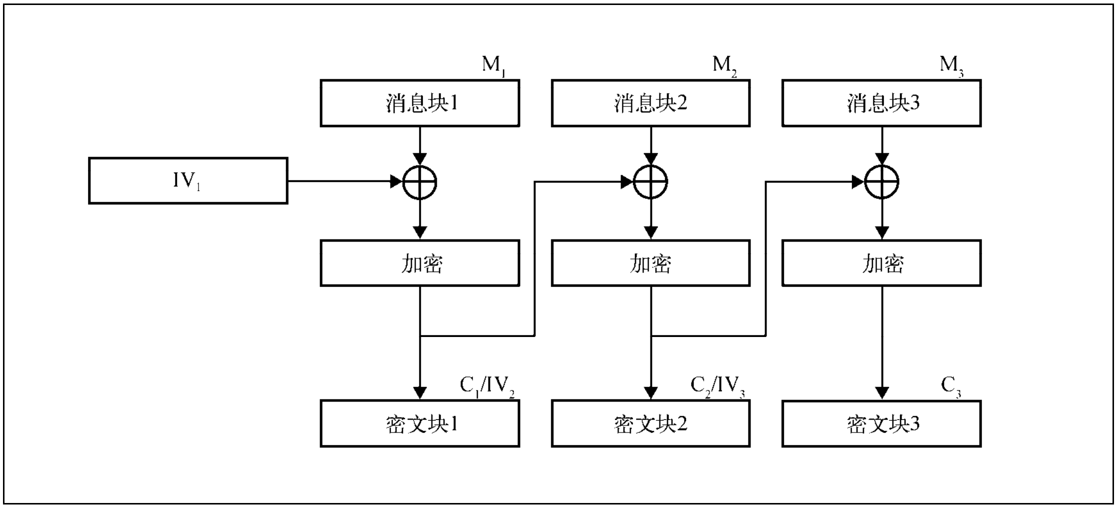


图1-4　CBC模式加密

非对称加密（asymmetric encryption）又称为公钥加密（public key cryptography），它是另一种方法，使用两个密钥，而不是一个；其中一个密钥是私密的，另一个是公开的。顾名思义，一个密钥用于私人，另一个密钥将会被所有人共享。这两个密钥之间存在一些特殊的数学关系，使得密钥具备一些有用的特性。如果你利用某人的公钥加密数据，那么只有他们对应的私钥能够解密，如图1-5所示。从另一个方面讲，如果某人用私钥加密数据，任何人都可以利用对应的公钥解开消息。后面这种操作不提供机密性，但可以用作数字签名。

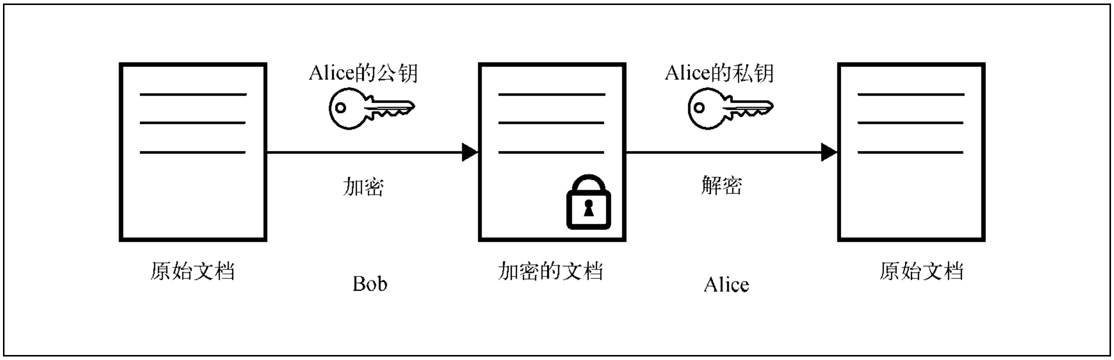


图1-5　非对称加密

非对称加密使得大规模团体的安全通信大幅简化。假设你可以广泛并且安全地分享你的公钥（PKI的工作将会在第3章中进行讨论），那么任何人都可以向你发送消息，只有你可以阅读。如果他们使用各自的私钥签名，你还可以精确地知道消息出自何人之手。

虽然公钥密码的属性非常有趣，但它却非常缓慢，不适用于数据量大的场景。因此，它往往被部署于进行身份验证和共享秘密的协商，这些秘密后续将用于快速的对称加密。

RSA（得名于三个人的姓氏首字母：Ron Rivest、Adi Shamir和Leonard Adleman）是目前最普遍部署的非对称加密算法[①]。现在推荐的RSA强度是2048位，强度等同于112位的对称密钥。我将会在本章稍后更加详细地讨论密码强度。

6. 数字签名

数字签名（digital signature）是一个密码学方案。它使得验证一条电子消息或者一篇电子文档的真实性成为可能。前面描述的MAC就是一种电子签名，它可以利用事先安全交换的散列密钥验证真实性。虽然这种校验非常有用，但仍有不足，因为它仍然依赖于一个私有密钥。

借助公钥密码，数字签名可以与现实生活中的手写签名类似。我们可以利用公钥密码的非对称性设计出一种算法，使用私钥对消息进行签名，并使用对应的公钥验证它。

实际的方式依照选择的公钥密码体系而有所不同。下面以RSA为例。RSA可以用于加密，也可以用于解密。如果使用RSA私钥加密，那么仅能通过对应的公钥解密。我们可以利用这个性质，并且结合散列函数，实现数字签名。

(1) 计算希望签名的文档的散列。不论输入文档的长度如何，输出长度总是固定的。比如，使用SHA256就是256位。

(2) 对结果散列和一些额外的元数据进行编码。比如，接收方需要知道你使用的散列算法，否则不能处理签名。

(3) 使用私钥加密编码过的数据，其结果就是签名，可以追加到文档中作为身份验证的依据。

为了验证签名，接收方接收文档并使用相同的散列算法独立计算文档散列。接着，她使用公钥对消息进行解密，将散列解码出来，再确认使用的散列算法是否正确，解密出的散列是否与本地计算的相同。这个方案的强度取决于加密、散列以及编码组件各自的强度。

> **注意**
>
> 并非所有的数字签名算法都与RSA的工作方式一致。事实上，RSA是一个特例，因为它可以同时用于加密和数字签名。其他流行的公钥密码算法则不能用于加密，比如DSA和ECDSA，它们依赖其他方式进行签名。

7. 随机数生成

在密码学中，所有的安全性都依赖于生成随机数的质量。在本章中，你已经看到，安全性构建于已知的算法和未知的密钥之上；而密钥最简单的形式就是非常长的随机数。

之所以说随机数不易生成，是因为计算机是十分善于预测的，它们会严格按照指令执行。如

---

① RSA，https://en.wikipedia.org/wiki/RSA_%28cryptosystem%29（维基百科，检索于2014年6月2日）。

果告诉它生成一个随机数，它很可能做不好这项工作[①]。真正的随机数只能通过观测特定的物理处理器才能得到。没有的话，计算机将关注于收集少量的熵（entropy）。这通常意味着监视按键状态、鼠标移动，以及各种外设（比如硬盘）的交互情况。

通过这种方式收集熵是一种*真随机数生成器*（true random number generator，TRNG），但是直接使用这种方式并不足够可靠。打个比方，你可能需要生成一个4096位的密钥，但是系统可能只有数百位的熵可用。如果没有可靠的外部事件可以收集到足够的熵，系统就可能会停止。

基于上面的原因，我们在实际使用中依靠的是*伪随机数生成器*（pseudorandom number generator，PRNG）。当然，PRNG也要利用少量真正的随机数使系统运转起来。这个过程被称为*种子设定*（seeding）。利用种子，PRNG根据需要构造出无限数量的伪随机数。普通用途的PRNG被常常用于编程，但它们并不适用于密码学，尽管其输出看起来就是随机的。*加密安全伪随机数生成器*（cryptographically secure pseudorandom number generator，CPRNG）是不可预测的PRNG。这个性质对安全来说非常关键，一定不能让攻击者对观察到的CPRNG输出进行内部状态的逆向工程。

1.4.2　协议

加密基元本身其实没什么用，诸如加密和散列算法。我们只有将这些元素组合成方案和协议，才能满足复杂的安全需求。为了说明我们需要怎么做，先来看一个简化的密码协议，这个协议可以使Alice和Bob安全地通信。我们的目标是全部三个重要需求：机密性、完整性和真实性。

我们假设协议允许交换任意数量的消息。因为对称加密擅长对大量数据进行加密，所以选取我们最喜欢的AES算法来进行数据加密。使用AES，Alice和Bob可以安全地交换消息，Mallory看不到他们通信的内容。但是这还不够，因为Mallory还可以干其他事情，比如，神不知鬼不觉地修改消息。为了解决这个问题，我们使用只有Alice和Bob知道的散列密钥计算每个消息的MAC。在发送消息的同时，也发送消息的MAC。

现在，Mallory再也不能修改消息了。然而，她仍然可以丢弃或者重发任意消息。为了解决这个问题，我们扩展协议，为每条消息指定序号。最为重要的是，我们将序号作为MAC计算数据的一部分。如果发现序号出现空缺，就能知道消息丢了。如果我们发现序号出现重复，就检测重放攻击。为了得到最佳结果，我们应使用某个特殊消息来标记会话结束。如果没有这个消息，Mallory能够悄悄地结束（截断）会话。

如果所有措施都已到位，Mallory最多只能做到阻止Alice和Bob与其他人进行通信。我们对此无能为力。

到目前为止，一切都好，但是我们仍然有一大块缺失：Alice和Bob如何协商得到需要的两个密钥（一个用于加密，一个用于完整性验证），同时还要当心Mallory？我们通过为协议增加两个额外的步骤来解决这个问题。

---

① 一些新型处理器内建了适于加密使用的随机数产生器。也有一些专业外设（比如，以闪存盘的形式）可以为操作系统提供额外的熵。

首先，在会话的开始，我们使用公钥密码对会话双方进行身份验证。举个例子，Alice生成一个随机数，并要求Bob对其签名以证明真的是他。Bob也要求Alice做相同的事情。

除了身份验证之外，我们还可以使用密钥交换方案对加密密钥进行秘密协商。继续举例，Alice可以生成所有密钥，用Bob的公钥加密，再发送给Bob，这就是RSA密钥交换的工作方式。我们也可以使用Diffie-Hellman（DH）密钥交换协议作为替代。后者相对速度更慢，但提供了更多的安全特性。

最后，我们的协议完工时的状态是：(1) 以握手阶段开始，包括身份验证和密钥交换；(2) 接下来是数据交换阶段，保证机密性和完整性；(3) 以关闭序列结束。站在宏观的角度来看，我们的协议与SSL和TLS完成的工作相似。

1.4.3　攻击密码

复杂系统往往会受到多种方式的攻击，密码系统也不例外。首先，你可以攻击加密基元本身。如果密钥很短，攻击者可以暴力破解。这种攻击通常需要相当多的运算能力和时间。对攻击者来说，如果系统使用的基元存在已知缺陷，他就可以使用解析攻击，从而更简单、更快地达成攻击目标。

人们一般都能很好地理解加密基元，因为它们相对直接，并且只完成一件工作。整体方案往往更容易遭受攻击，因为它们引入了额外的复杂性。在某些场景下，即使是密码学家也会争论执行特定操作的正确方法；但不论是基元还是方案，都比协议更安全。因为协议引入了更多的复杂性，并且攻击界面也大得多。

此外，也存在针对协议实现（implementation）的攻击；换言之，就是利用软件的bug。比如，绝大多数密码库都使用C甚至汇编这样的低级语言编写（出于性能的原因），非常容易引入灾难性的编程错误。即便没有这些bug，要实现基元、方案和协议，保证它们不被滥用，也需要很高的技巧。举个例子，某些算法的本地实现可以被计时攻击（timing attack）所利用，攻击者可以通过观察特定操作执行的时间破解加密。

有一种现象也非常普遍，那就是没有密码经验的程序员企图实现（甚至设计）加密协议和方案，理所应当地造成了不安全的结果。

所以，我们通常会说加密被绕过，而不是被攻击。这句话意味着使用的基元都很坚实，但软件体系不牢固。再进一步说，密钥是非常诱人的攻击目标：如果我们可以更轻松地闯进服务器拿到密钥，为什么还要花费数月时间暴力破解它？许多失败的加密案例都可以参照下面的简单规则避免：(1) 使用完善的协议，不要自己设计；(2) 使用高级库，避免直接操作加密；(3) 使用完备的基元，辅以足够强壮的密钥长度。

1.4.4　衡量强度

我们使用攻破某个基元所需执行的操作数量衡量密码系统的强度，以安全位数来表示。最容易做到的符合正确部署的行为就是部署长度足够强的密码，而且规则很简单：大部分系统部署128位（2^{128}次操作）就足够了；如果需要长期的安全或者较大的安全宽限期，则使用256位。

> **注意**
> 对称密码系统的强度按位数增加以几何基数增长，这意味着密码增加1位，强度翻一倍。

实际使用的情况更复杂一些，因为并非所有操作的安全性都能同等度量。所以，对于对称加密的操作、非对称加密的操作、椭圆曲线加密算法以及其他操作，我们使用不同的位数。你可以使用表1-2中的信息将一种大小转换为另一种大小。

表1-2 改编自ECRYPT2（2012）的安全级别和对应的安全强度位数

编号	保护	对称	非对称	DH	椭圆曲线	散列
1	个人实时攻击	32	–	–	–	–
2	对抗小规模组织非常短期的保护	64	816	816	128	128
3	对抗中等规模组织的短期保护	72	1008	1008	144	144
4	对抗专业代理机构非常短期的保护	80	1248	1248	160	160
5	短期保护（10年）	96	1776	1776	192	192
6	中期保护（20年）	112	2432	2432	224	224
7	长期保护（30年）	128	3248	3248	256	256
8	长期保护，增加对量子计算机的防御能力	256	15 424	15 424	512	512

这份数据是我从2012年的一份关于密钥和算法强度的报告[①]中节选出来的。它不仅粗略地展示了不同类型算法的位数对应关系，而且根据攻击者的能力和保护的时间长度两个方面定义了强度。尽管我们对安全与否的讨论都倾向于假设在现在这个时间点，但实际上安全性是一个与时间相关的函数。加密的强度会随时间发生变化，这是因为随着时间的推移，计算机会更快、更便宜。安全性同时也是与资源相关的函数。一个短密钥对个人来说可能不可破解，但专业代理机构就可以达成破解目标。因此，当我们讨论安全性时，提出诸如“针对谁的安全”和“多长时间的安全”这种问题会更有价值。

> **注意**
> 因为无法做到精确衡量密码强度，所以你可以找到各种不同的推荐。它们大多数非常相似，只有一点点区别。以我的经验，ENISA（European Union Agency for Network and Information Security，欧洲网络与信息安全联合机构）提供了高级文档，可以以多种层次[②]给大家清晰的指导[③]。如果需要查阅和比较其他推荐文档，请访问keylength.com[④]。

尽管表1-2提供了非常有用的信息，但你可能会发现它们难以使用，因为那些值与我们通常

① ECRYPT2 Yearly Report on Algorithms and Keysizes，http://www.ecrypt.eu.org/（European Network of Excellence for Cryptology II，2012年9月30日）。

② Recommended cryptographic measures - Securing personal data，https://www.enisa.europa.eu/activities/identity-and-trust/library/deliverables/recommended-cryptographic-measures-securing-personal-data（ENISA，2013年11月4日）。

③ Algorithms, Key Sizes and Parameters Report，https://www.enisa.europa.eu/activities/identity-and-trust/library/deliverables/algorithms-key-size-and-parameters-report-2014（ENISA，2013年10月29日）。

④ BlueKrypt: Cryptographic Key Length Recommendation，http://www.keylength.com/（BlueKrypt，检索于2014年6月4日）。

使用的密钥长度不对应。在实践中，你会发现表1-3在将一组安全位转换成另一组时更有用[①]。

表1-3　常用密钥长度的加密强度映射

对　称	RSA/DSA/DH	椭圆曲线	散　列
80	1024	160	160
112	2048	224	224
128	3072	256	256
256	15 360	512	512

1.4.5　中间人攻击

针对传输层安全性的攻击绝大多数来自中间人（man-in-the-middle，MITM）攻击。这意味着除了会话双方的两个团体，还存在一个恶意团体。如果攻击者只是监听双方的会话，我们称之为被动网络攻击（passive network attack）。如果攻击者主动改变数据流或者影响双方会话，我们则称之为主动网络攻击（active network attack）。参见图1-6。

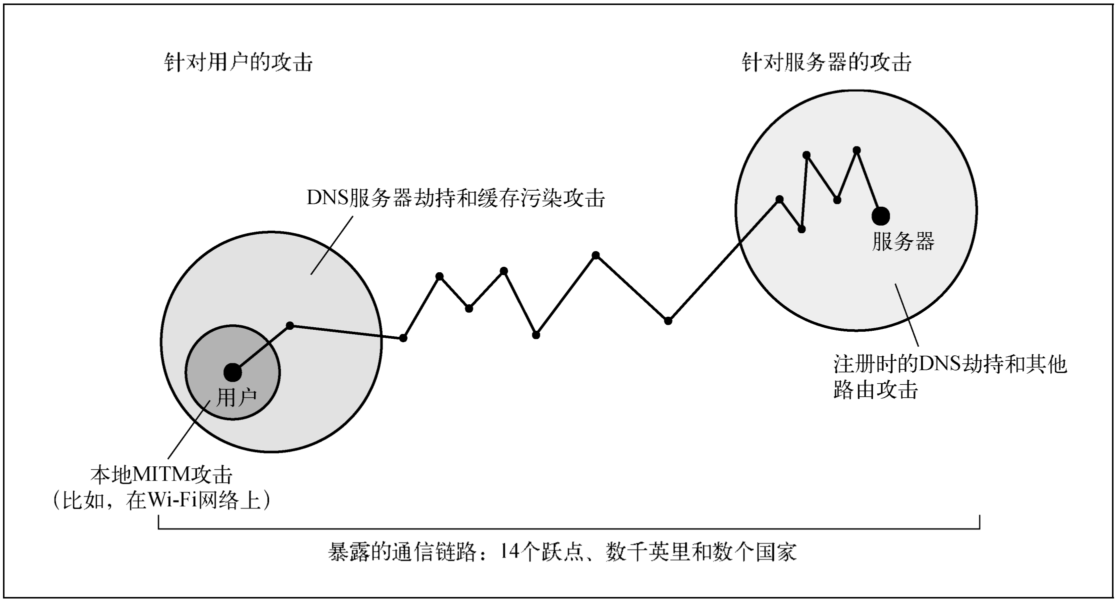


图1-6　SSL/TLS威胁的概念模型

1. 取得访问权

在很多案例中，攻击者需要接近受害人或服务器，或者取得通信设施的访问权。无论是谁，只要能进入线路和中间通信节点（比如路由器），就能够看到线路上通行的数据帧，并且能够对

① NIST Special Publication 800-57: Recommendation for Key Management – Part 1: General, Revision 3，http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf（NIST，2012年7月）。

它们进行干预。可以通过割开电缆[①]、与运营商共谋[②]或者直接侵入设备[③]来获取访问权。

理论上，执行MITM攻击的最简单方法是加入网络，然后将受害者的通信重新路由到恶意节点。现在很多人都在使用的无线网络并没有身份验证机制，任何人都可以加入，所以尤其容易受到这种攻击。

其他攻击方式包括妨碍域名解析、IP地址路由等的路由基础设施。

- ARP欺骗

地址解析协议（address resolution protocol，ARP）用于在局域网中将MAC地址[④]与IP地址进行关联。进入网络的攻击者可以声明任何IP地址，并对网络流量进行有效的重路由。

- WPAD劫持

浏览器使用Web代理自动发现协议（web proxy auto-discovery protocol，WPAD）自动获取HTTP代理的配置。WPAD使用了好几种方法，包括DHCP和DNS。为了攻击WPAD，攻击者在局域网中启动一台服务器并将其通知到那些寻找服务的本地客户端。

- DNS劫持

只要攻击者能通过注册或者改变DNS配置来劫持某个域名，就可以劫持访问这个域名的所有流量。

- DNS缓存中毒

DNS缓存中毒（DNS cache poisoning）是一种攻击者利用DNS缓存服务器的缺陷在缓存中注入非法域名信息的攻击方式。成功完成这种攻击以后，受影响的DNS服务器的所有用户都将收到攻击者构造的非法信息。

- BGP路由劫持

边界网关协议（border gateway protocol，BGP）是一种互联网骨干网络用于发现如何精确定位IP地址段的路由协议。如果某个非法路由信息被一个或更多路由器所接受，所有通往某个特定IP地址段的流量都将被重定向到另一处，即攻击者那里。

2. 被动攻击

被动攻击对于未加密的流量最为有用。整个2013年，世界各国的政府机构对互联网流量的常规监控和大量存储变得人尽皆知。例如，有人宣称英国谍报部门GCHQ记录了英国所有的互联网流量并保存三天时间[⑤]。你的电子邮件消息、照片、互联网聊天记录，以及其他数据都存放在某

---

① The Creepy, Long-Standing Practice of Undersea Cable Tapping，http://www.theatlantic.com/international/archive/2013/07/the-creepy-long-standing-practice-of-undersea-cable-tapping/277855/（《大西洋月刊》，2013年7月16日）。

② New Details About NSA’s Collaborative Relationships With America's Biggest Telecom Companies From Snowden Docs，http://www.matthewaid.com/post/59765378513/new-details-about-nsas-collaborative（《华盛顿邮报》，2013年8月30日）。

③ Photos of an NSA “upgrade” factory show Cisco router getting implant，http://arstechnica.com/tech-policy/2014/05/photos-of-an-nsa-upgrade-factory-show-cisco-router-getting-implant/（Ars Technica，2014年5月14日）。

④ 这种情况下，MAC代表媒体接入控制。它是生产厂家指定给网卡的唯一标识。

⑤ GCHQ taps fibre-optic cables for secret access to world’s communications，http://www.theguardian.com/uk/2013/jun/21/gchq-cables-secret-world-communications-nsa（《卫报》，2013年6月21日）。

处的一个数据库中，等待着进行相互参照，分析其是否具有特殊目的。如果大量数据都像这样处理，那么可以想象某些数据会存储更长时间甚至无限期存放。为了应对这些，IETF宣布“渗透监听就是攻击”[①]，需要尽可能使用加密进行防御。

即使针对加密数据，被动攻击仍可以作为全局策略的一个要素发挥作用。比如，可以先保存抓取的加密数据，直到破解加密之后再使用。当前困难的事情在十年以后可能就会变得简单，因为随着时间的推移，计算机会越来越强大，越来越便宜，也有可能发现加密基元的弱点。

让事情变得更为糟糕的是，计算机系统通常存在一个严重的配置缺陷，使得攻击者可以追溯解密记录的数据。TLS中最常见的密钥交换算法是基于RSA算法的；在使用这种算法的系统中，密钥交换使用的RSA密钥也用于解密过去所有的会话。其他密钥交换算法不存在这个问题，被称为支持前向保密（forward secrecy）。不幸的是，大部分系统使用的仍然是RSA算法。举个例子，爱德华·斯诺登使用的加密邮件系统Lavabit就不支持前向保密。FBI拿着法庭的传票，迫使Lavabit交出了他们的加密密钥[②]。有了密钥，FBI就能解开有记录的通信（当然，只要他们记录过），于是爱德华·斯诺登的邮件就被破译了。

被动攻击的效果非常好，这既是因为还有非常多的数据未被加密，也是因为大量收集信息的过程可以全自动化。截至2014年7月，到达Gmail的邮件只有58%进行过加密[③]，这可以很好地说明现在的状况。

3. 主动攻击

当大家谈论MITM攻击时，指的基本上都是主动攻击。Mallory可以利用主动攻击以某种方式干预通信。传统的MITM会攻击目标的身份验证系统，诱使Alice认为她正在与Bob通信。如果攻击成功，Mallory会收到Alice的消息并转给Bob。虽然Alice发送消息时会进行加密，但这并不是问题。因为她的发送对象是Mallory，后者可以用她与Alice协商的密钥解密消息。

在TLS方面，对Mallory来说，理想的场景是Alice认为她给出的证书是有效的并且接受证书。那样的话，攻击天衣无缝，基本上无法被发现[④]。要使证书有效，需要攻击者扮演公钥证书体系的角色。近年来，已经有很多这类攻击的案例了，我在第4章中记录了那些众所周知的案例。利用验证代码中的bug，攻击者可以构造看似有效的证书。历史上，这个领域内的bug较为常见，我会在第6章中讨论其中一些示例。如果前面所有的攻击最终都失败了，Mallory就会发送一个无效证书碰碰运气，看Alice会不会强行无视证书警告。几年前这种情况在叙利亚发生过[⑤]。

作为强大的应用发布平台，浏览器的势头不断上升，也为主动攻击提供了新的攻击向量。在

---

① RFC 7258: Pervasive Monitoring Is an Attack，http://tools.ietf.org/html/rfc7258（S. Farrell和H. Tschofenig，2014年5月）。

② Lavabit，https://en.wikipedia.org/wiki/Lavabit（维基百科，检索于2014年6月4日）。

③ Transparency Report: Email encryption in transit，http://www.google.com/transparencyreport/saferemail/（Google Gmail，检索于2014提7月27日）。

④ 除非你非常执着地亲自跟踪所有遇见过的证书，某些浏览器扩展可以做到（比如Certificate Patrol for Firefox）。

⑤ A Syrian Man-In-The-Middle Attack against Facebook，https://www.eff.org/deeplinks/2011/05/syrian-man-middle-against-facebook（The Electronic Frontier Foundation，2011年5月5日）。

这个场景中，身份验证不会受到攻击，攻击者操纵受害者的浏览器，使浏览器提交精心构造的请求从而破坏加密。近年来，这些攻击向量已经被用作新的攻击手段，对TLS进行了攻击。大家可以在第7章中找到更多内容。

主动攻击的攻击力非常强大，但更难扩展。与被动攻击仅仅将观察的分组进行复制（这种操作非常简单）相比，主动攻击需要进行更多处理并投入更多努力来跟踪个体连接。所以主动攻击对软件和硬件的需求会多很多。重定向大流量也容易被发现。同样，大规则证书欺诈攻击也很难奏效，原因是应付这么多个人和组织对各种网站证书的跟踪监测非常困难。最有希望成功的攻击方式是利用执行中的漏洞，绕过身份验证。但是，像这样能带来毁灭性结果的程序错误较为罕见。

基于上述原因，主动攻击最可能用于攻击高价值的个人目标。主动攻击无法实现自动化，这意味着它们需要额外的工作，消耗很大，也因此更难以证明其价值。

有迹象表明NSA在一个名为QuantumInsert[①]的项目中部署了一套庞大的设施，可以用来攻击互联网上的几乎任意主机。

这个项目是MITM方案的变种。它不是针对目标的加密系统，而是利用选定对象个人的浏览器漏洞投递攻击。NSA在通信设施中放置一些特殊的分组注入节点，从而得到比真实服务器更快的连接请求响应能力，并使它可以将部分流量重定向到漏洞利用的服务器。

① Attacking Tor: How the NSA Targets Users' Online Anonymity，https://www.schneier.com/essays/archives/2013/10/attacking_tor_how_th.html（Bruce Schneier，2013年10月4日）。

第2章

协　议

2

TLS是一种密码学协议，用于保证两个团体之间的会话安全。会话是由任意数量的消息组成的。本章将讨论最新的协议版本TLS 1.2，也会在合适的时候对之前的协议版本进行简要说明。

我的目标是进行宏观概述，让你能够理解其工作原理，而不会因具体的实现细节分心。我尽可能使用消息内容的示例，避免只给出定义。本章中定义用到的语法在本质上与TLS规范完全一致，仅有稍微简化。要得到更多有关语法和完整协议的参考，请从RFC 5246开始了解，它描述了TLS 1.2的规范[1]，但是并没有完整的细节。还有许多其他的相关RFC，本章会逐一引用到。

了解TLS的最好方式是观察现实中的网络流量。我最喜欢的方法是使用网络捕获工具Wireshark[2]，它带有一个TLS协议分析器：用你喜爱的浏览器进入一个安全网站，再启用Wireshark监视连接（最好限制只捕捉一个主机名和端口443），并观察协议消息。

在理解TLS以后（不用努力了解一切；因为特性太多，理解所有特性非常困难），你可以自由浏览各类RFC，甚至在关键邮件列表中"潜水"。我最喜欢的两处是TLS工作组文档页[3]和TLS工作组邮件列表[4]。你可以在前者中找到关键文档和新提案的清单，可以在后者中跟进有关TLS未来发展方向的讨论。

2.1　记录协议

宏观上，TLS以*记录协议*（record protocol）实现。记录协议负责在传输连接上交换的所有底层消息，并可以配置加密。每一条TLS记录以一个短标头起始。标头包含记录内容的类型（或子协议）、协议版本和长度。消息数据紧跟在标头之后，如图2-1所示。

---

① RFC 5246: The Transport Layer Security Protocol Version 1.2，http://tools.ietf.org/html/rfc5246（T. Dierks和E. Rescorla，2008年8月）。

② Wireshark，https://www.wireshark.org（检索于2015年7月13日）。

③ TLS working group documents，https://datatracker.ietf.org/wg/tls/documents/（IETF，检索于2014年7月19日）。

④ TLS working group mailing list archives，http://www.ietf.org/mail-archive/web/tls/current/（IETF，检索于2014年7月19日）。

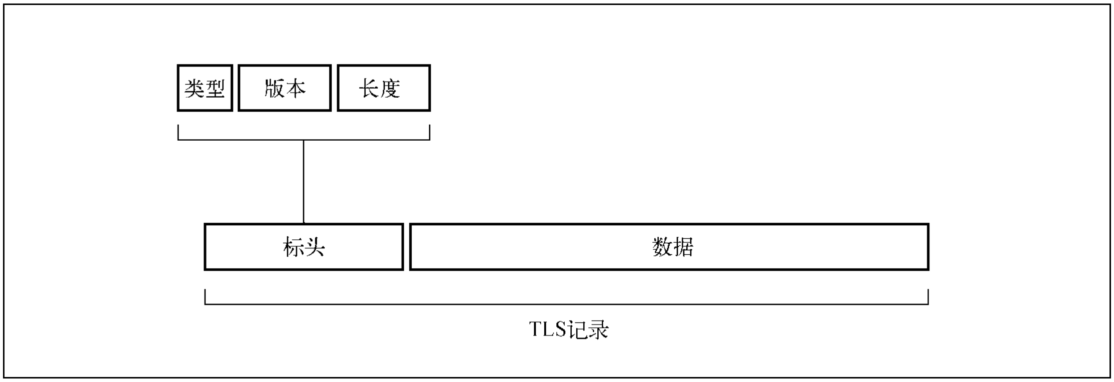


图2-1　TLS记录

可以更为正式地将TLS记录的字段定义为如下所示。

```
struct {
    uint8 major;
    uint8 minor;
} ProtocolVersion;

enum {
    change_cipher_spec (20),
    alert (21),
    handshake (22),
    application_data (23)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length; /* 最大长度为2^14 (16 384) 字节 */
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

除了这些可见的字段，还会给每一个TLS记录指定唯一的64位序列号，但不会在线路上传输。任一端都有自身的序列号并跟踪来自另一端记录的数量。这些值是对抗重放攻击的一部分。稍后你可以看到它的工作原理。

记录协议从多个重要的宏观角度对通信进行考量，是一个很有用的协议抽象。

- 消息传输

  记录协议传输由其他协议层提交给它的不透明数据缓冲区。如果缓冲区超过记录的长度限制（16 384字节），记录协议会将其切分成更小的片段。反过来也是可能的，属于同一个子协议的小缓冲区也可以组合成一个单独的记录。

- 加密以及完整性验证

  在一个刚建立起来的连接上，最初的消息传输没有受到任何保护（从技术上讲，就是使用了`TLS_NULL_WITH_NULL_NULL`密码套件）。这是必需的，否则第一次协商就无法进行。但

是，一旦握手完成，记录层就开始按照协商取得的连接参数进行加密和完整性验证[①]。

- 压缩

  理论上，在加密之前透明地对数据进行压缩非常好，可是实践中几乎没有人这样做。这主要是因为每个应用在HTTP层就已经对它们的出口流量进行过压缩。这个特性在2012年遭受过一次严重打击，当时CRIME攻击使压缩成为不安全的特性[②]，所以现在也不会再被使用。

- 扩展性

  记录协议只关注数据传输和加密，而将所有其他特性转交给子协议。这个方法使TLS可以扩展，因为可以很方便地添加子协议。伴随着记录协议而被加密，所有子协议都会以协商取得的连接参数自动得到保护。

TLS的主规格说明书定义了四个核心子协议：握手协议（handshake protocol）、密钥规格变更协议（change cipher spec protocol）、应用数据协议（application data protocol）和警报协议（alert protocol）。

2.2　握手协议

握手是TLS协议中最精密复杂的部分。在这个过程中，通信双方协商连接参数，并且完成身份验证。根据使用的功能的不同，整个过程通常需要交换6~10条消息。根据配置和支持的协议扩展的不同，交换过程可能有许多变种。在使用中经常可以观察到以下三种流程：(1) 完整的握手，对服务器进行身份验证；(2) 恢复之前的会话采用的简短握手；(3) 对客户端和服务器都进行身份验证的握手。

握手协议消息的标头信息包含消息类型（1字节）和长度（3字节），余下的信息则取决于消息类型：

```
struct {
    HandshakeType msg_type;
    uint24 length;
    HandshakeMessage message;
} Handshake;
```

2.2.1　完整的握手

每一个TLS连接都会以握手开始。如果客户端此前并未与服务器建立会话，那么双方会执行一次完整的握手流程来协商TLS会话。握手过程中，客户端和服务器将进行以下四个主要步骤。

(1) 交换各自支持的功能，对需要的连接参数达成一致。

(2) 验证出示的证书，或使用其他方式进行身份验证。

---

① 大多数情况下，这代表后续流量是加密的并且其完整性已进行过验证。但也存在少量套件未使用加密而仅使用了完整性校验的情况。

② 我会在7.3节中讨论CRIME攻击以及各种其他压缩相关的弱点。

(3) 对将用于保护会话的共享主密钥达成一致。
(4) 验证握手消息并未被第三方团体修改。

> **注意**
> 在实际使用中，第2步和第3步都是密钥交换（更通用的说法是密钥生成）的一部分，密钥交换是一个单独的步骤。我更喜欢将它们分开来说，用以强调协议的安全性取决于正确的身份验证。身份验证有效地在TLS的外层工作。如果没有身份验证，主动攻击者就可以将自身嵌入会话，并冒充会话的另一端。

本节会讨论最常见的TLS握手流程，就是一种在不需要身份验证的客户端与需要身份验证的服务器之间的握手，如图2-2所示。后面几节将介绍其他的流程：客户端身份验证和会话恢复。

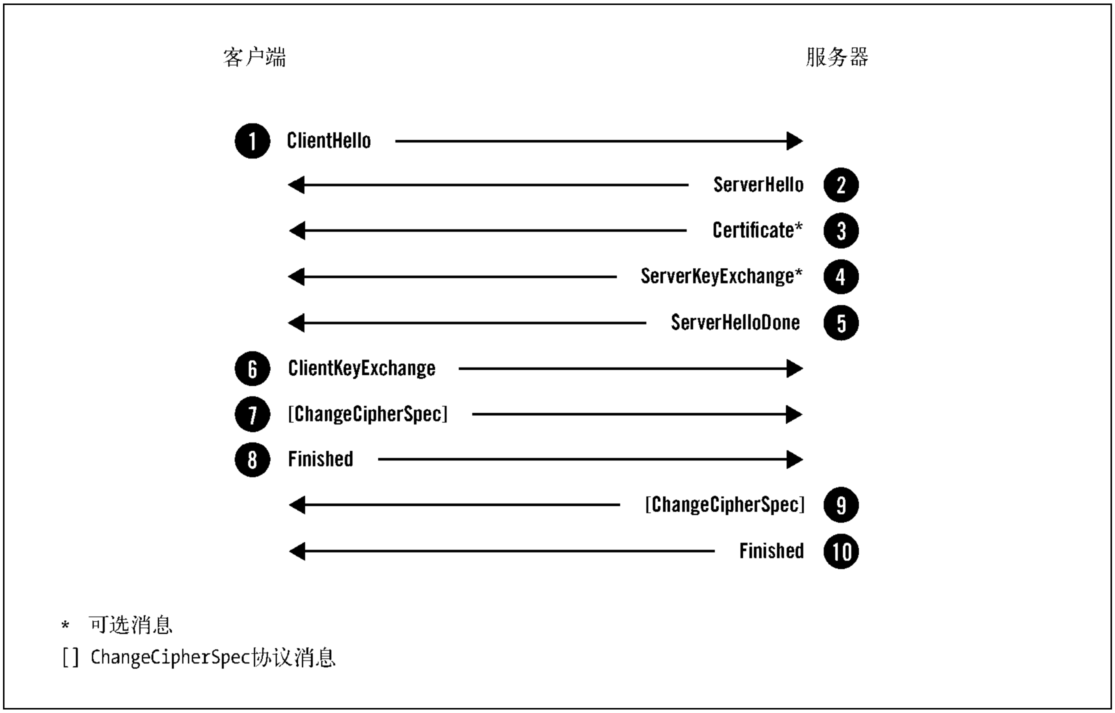


图2-2　对服务器进行身份验证的完整握手

(1) 客户端开始新的握手，并将自身支持的功能提交给服务器。
(2) 服务器选择连接参数。
(3) 服务器发送其证书链（仅当需要服务器身份验证时）。
(4) 根据选择的密钥交换方式，服务器发送生成主密钥的额外信息。
(5) 服务器通知自己完成了协商过程。
(6) 客户端发送生成主密钥所需的额外信息。

(7) 客户端切换加密方式并通知服务器。

(8) 客户端计算发送和接收到的握手消息的MAC并发送。

(9) 服务器切换加密方式并通知客户端。

(10) 服务器计算发送和接收到的握手消息的MAC并发送。

假设没有出现错误，到这一步，连接就建立起来了，可以开始发送应用数据。现在让我们了解一下这些握手消息的更多细节。

1. ClientHello

在一次新的握手流程中，ClientHello消息总是第一条消息。这条消息将客户端的功能和首选项传送给服务器。客户端会在新建连接后，希望重新协商或者响应服务器发起的重新协商请求（由HelloRequest消息指示）时，发送这条消息。

在下面的例子中，你可以观察到ClientHello消息。为了更简洁，我减少了一些信息展示，但是包含了所有的关键元素。

```
Handshake protocol: ClientHello
    Version: TLS 1.2
    Random
        Client time: May 22, 2030 02:43:46 GMT
        Random bytes: b76b0e61829557eb4c611adfd2d36eb232dc1332fe29802e321ee871
    Session ID: (empty)
    Cipher Suites
        Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
        Suite: TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
        Suite: TLS_RSA_WITH_AES_128_GCM_SHA256
        Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
        Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA
        Suite: TLS_RSA_WITH_AES_128_CBC_SHA
        Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA
        Suite: TLS_RSA_WITH_RC4_128_SHA
    Compression methods
        Method: null
    Extensions
        Extension: server_name
            Hostname: www.feistyduck.com
        Extension: renegotiation_info
        Extension: elliptic_curves
            Named curve: secp256r1
            Named curve: secp384r1
        Extension: signature_algorithms
            Algorithm: sha1/rsa
            Algorithm: sha256/rsa
            Algorithm: sha1/ecdsa
            Algorithm: sha256/ecdsa
```

可以看到，绝大多数消息字段光看名称就很容易理解，而且消息的结构也很容易理解。

- Version

  协议版本（protocol version）指示客户端支持的最佳协议版本。

- Random

随机数（random）字段包含32字节的数据。当然，只有28字节是随机生成的；剩余的4字节包含额外的信息，受客户端时钟的影响。准确来说，客户端时间与协议不相关，而且协议规格文档中言及此事时也很清楚（“基本的TLS协议不需要正确设置时钟，更高层或应用协议可以定义额外的需求项。”）；该字段是1994年在Netscape Navigator中发现了一个严重故障之后，为了防御弱随机数生成器而引入的[①]。尽管这个字段曾经一直含有精确时间的部分，但现在仍然有人担心客户端时间可能被用于大规模浏览器指纹采集[②]，所以一些浏览器会给它们的时间添加时钟扭曲（正如你在示例中所看到的那样），或者简单地发送随机的4字节。

在握手时，客户端和服务器都会提供随机数。这种随机性对每次握手都是独一无二的，在身份验证中起着举足轻重的作用。它可以防止重放攻击，并确认初始数据交换的完整性。

- `Session ID`

  在第一次连接时，会话ID（session ID）字段是空的，这表示客户端并不希望恢复某个已存在的会话。在后续的连接中，这个字段可以保存会话的唯一标识。服务器可以借助会话ID在自己的缓存中找到对应的会话状态。典型的会话ID包含32字节随机生成的数据，这些数据本身并没有什么价值。

- `Cipher Suites`

  密码套件（cipher suite）块是由客户端支持的所有密码套件组成的列表，该列表是按优先级顺序排列的。

- `Compression`

  客户端可以提交一个或多个支持压缩的方法。默认的压缩方法是`null`，代表没有压缩。

- `Extensions`

  扩展（extension）块由任意数量的扩展组成。这些扩展会携带额外数据。我会在本章后面对最常见的扩展进行讨论。

2. ServerHello

`ServerHello`消息的意义是将服务器选择的连接参数传送回客户端。这个消息的结构与`ClientHello`类似，只是每个字段只包含一个选项。

```
Handshake protocol: ServerHello
    Version: TLS 1.2
    Random
        Server time: Mar 10, 2059 02:35:57 GMT
        Random bytes: 8469b09b480c1978182ce1b59290487609f41132312ca22aacaf5012
    Session ID: 4cae75c91cf5adf55f93c9fb5dd36d19903b1182029af3d527b7a42ef1c32c80
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
    Compression method: null
    Extensions
```

① 如果想得到有关这个问题的更多信息，请参考6.2.1节。

② Deprecating gmt_unix_time in TLS，https://tools.ietf.org/id/draft-mathewson-no-gmtunixtime-00.txt（N. Mathewson和B. Laurie，2013年12月）。

```
Extension: server_name
Extension: renegotiation_info
```

服务器无需支持客户端支持的最佳版本。如果服务器不支持与客户端相同的版本，可以提供某个其他版本以期待客户端能够接受。

2

3. Certificate

典型的Certificate消息用于携带服务器X.509证书链。证书链是以ASN.1 DER编码的一系列证书，一个接着一个组合而成。主证书必须第一个发送，中间证书按照正确的顺序跟在主证书之后。根证书可以并且应该省略掉，因为在这个场景中它没有用处。

服务器必须保证它发送的证书与选择的算法套件一致。比方说，公钥算法与套件中使用的必须匹配。除此以外，一些密钥交换算法依赖嵌入证书的特定数据，而且要求证书必须以客户端支持的算法签名。所有这些都表明服务器需要配置多个证书（每个证书可能会配备不同的证书链）。

Certificate消息是可选的，因为并非所有套件都使用身份验证，也并非所有身份验证方法都需要证书。更进一步说，虽然消息默认使用X.509证书，但是也可以携带其他形式的标志；一些套件就依赖PGP密钥[1]。

4. ServerKeyExchange

ServerKeyExchange消息的目的是携带密钥交换的额外数据。消息内容对于不同的协商算法套件都会存在差异。在某些场景中，服务器不需要发送任何内容，这意味着在这些场景中根本不会发送ServerKeyExchange消息。

5. ServerHelloDone

ServerHelloDone消息表明服务器已经将所有预计的握手消息发送完毕。在此之后，服务器会等待客户端发送消息。

6. ClientKeyExchange

ClientKeyExchange消息携带客户端为密钥交换提供的所有信息。这个消息受协商的密码套件的影响，内容随着不同的协商密码套件而不同。

7. ChangeCipherSpec

ChangeCipherSpec消息表明发送端已取得用以生成连接参数的足够信息，已生成加密密钥，并且将切换到加密模式。客户端和服务器在条件成熟时都会发送这个消息。

> **注意**
>
> ChangeCipherSpec不属于握手消息，它是另一种协议，只有一条消息，作为它的子协议进行实现。这个设计的结果是这条消息不是握手完整性验证算法的一部分，这使得正确实现TLS更为困难。在2014年6月，人们发现OpenSSL对于ChangeCipherSpec消息

---

[1] RFC 5081: Using OpenPGP Keys for TLS Authentication，http://tools.ietf.org/html/rfc5081（N. Mavrogiannopoulos，2007年11月）。

> 的处理不正确，使得OpenSSL为主动网络攻击敞开了大门[①]。
> 同样的问题也出现在其他所有子协议中。主动网络攻击者利用缓冲机制在首次握手时发送未经验证的警报消息，更可以在开始加密以后破坏真正的警报消息[②]。为了避免更严重的问题，应用数据协议消息必须等到首次握手完成以后才能开始发送。

8. Finished

Finished消息意味着握手已经完成。消息内容将加密，以便双方可以安全地交换验证整个握手完整性所需的数据。

这个消息包含verify_data字段，它的值是握手过程中所有消息的散列值。这些消息在连接两端都按照各自所见的顺序排列，并以协商新得到的主密钥计算散列。这个过程是通过一个伪随机函数（pseudorandom function，PRF）来完成的，这个函数可以生成任意数量的伪随机数据。我将在本章的后续部分中对其进行介绍。散列函数与PRF一致，除非协商的套件指定使用其他算法。两端的计算方法一致，但会使用不同的标签：客户端使用client finished，而服务器则使用server finished。

```
verify_data = PRF(master_secret, finished_label, Hash(handshake_messages))
```

因为Finished消息是加密的，并且它们的完整性由协商MAC算法保证，所以主动网络攻击者不能改变握手消息并对vertify_data的值造假。

理论上攻击者也可以尝试找到一组伪造的握手消息，得到的值与真正消息计算出的verity_data的值完全一致。这种攻击本身就非常不容易，而且因为散列中混入了主密钥（攻击者不知道主密钥），所以攻击者根本不会尝试。

在TLS 1.2版本中，Finished消息的长度默认是12字节（96位），并且允许密码套件使用更长的长度。在此之前的版本，除了SSL 3使用36字节的定长消息，其他版本都使用12字节的定长消息。

2.2.2 客户端身份验证

尽管可以选择对任意一端进行身份验证，但人们几乎都启用了对服务器的身份验证。如果服务器选择的套件不是匿名的，那么就需要在Certificate消息中跟上自己的证书。

相比之下，服务器通过发送CertificateRequest消息请求对客户端进行身份验证。消息中列出所有可接受的客户端证书。作为响应，客户端发送自己的Certificate消息（使用与服务器发送证书相同的格式），并附上证书。此后，客户端发送CertificateVerify消息，证明自己拥有对应的私钥。完整的握手如图2-3所示。

---

① 你可以在6.1.1节中找到有关此缺陷的更多信息。

② The Alert attack，http://www.mitls.org/wsgi/alert-attack（miTLS，2012年2月）。

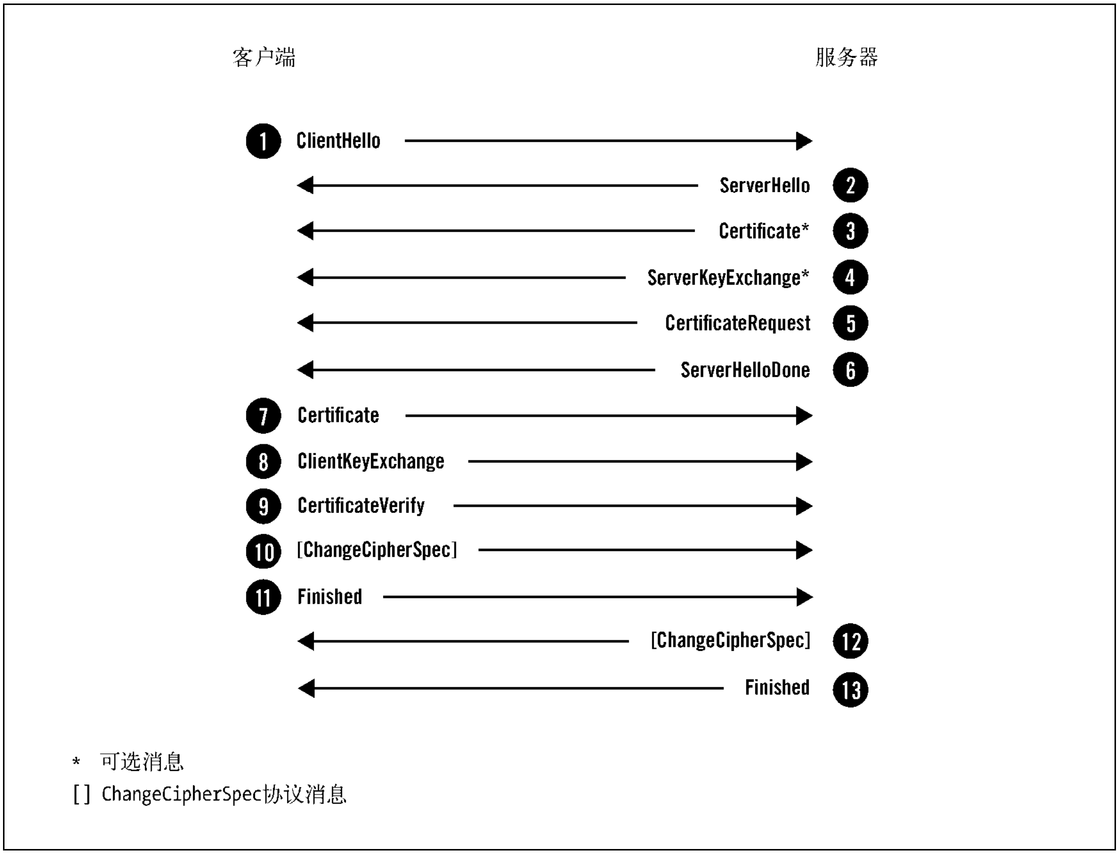


图2-3　完整的握手，在此期间客户端和服务器都会进行身份验证

只有已经过身份验证的服务器才被允许请求客户端身份验证。基于这个原因，这个选项被称为相互身份验证（mutual authentication）。

1. CertificateRequest

服务器使用CertificateRequest消息请求对客户端进行身份验证，并将其接受的证书的公钥和签名算法传送给客户端。它也可以选择发送一份自己接受的证书颁发机构列表，这些机构都用其可分辨名称来表示：

```
struct {
    ClientCertificateType certificate_types;
    SignatureAndHashAlgorithm supported_signature_algorithms;
    DistinguishedName certificate_authorities;
} CertificateRequest;
```

2. CertificateVerify

客户端使用CertificateVerify消息证明自己拥有的私钥与之前发送的客户端证书中的公钥

相对应。消息中包含一条到这一步为止的所有握手消息的签名：

```
struct {
    Signature handshake_messages_signature;
} CertificateVerify;
```

2.2.3　会话恢复

完整的握手协议非常复杂，需要很多握手消息和两次网络往返才能开始发送客户端应用数据。此外，握手执行的密钥学操作通常需要密集的CPU处理。身份验证通常以客户端和服务器证书验证（以及证书吊销检查）的形式完成，需要更多的工作。这其中的许多消耗都可以通过简短握手的方式节约下来。

最初的会话恢复机制是，在一次完整协商的连接断开时，客户端和服务器都会将会话的安全参数保存一段时间。希望使用会话恢复的服务器为会话指定唯一的标识，称为会话ID。服务器在ServerHello消息中将会话ID发回客户端（请参见2.2.2节中的示例）。

希望恢复早先会话的客户端将适当的会话ID放入ClientHello消息，然后提交。服务器如果愿意恢复会话，就将相同的会话ID放入ServerHello消息返回，接着使用之前协商的主密钥生成一套新的密钥，再切换到加密模式，发送Finished消息。客户端收到会话已恢复的消息以后，也进行相同的操作。这样的结果是握手只需要一次网络往返。简短握手如图2-4所示。

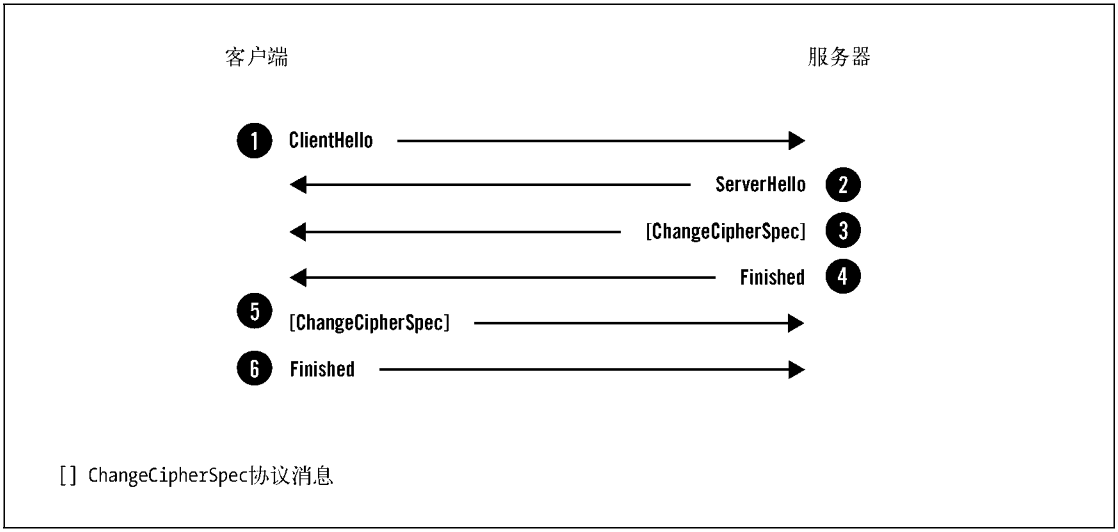


图2-4　简短握手，用于恢复已经建立的会话

用来替代服务器会话缓存和恢复的方案是使用*会话票证*（sesession ticket）。它是2006年引入的（参见RFC 4507），随后在2008年进行了更新（参见RFC 5077）。使用这种方式，除了所有的状态都保持在客户端（与HTTP Cookie的原理类似）之外，其消息流与服务器会话缓存是一样的。

2.3　密钥交换

密钥交换是握手过程中最引人入胜的部分。在TLS中，会话安全性取决于称为主密钥（master secret）的48字节共享密钥。密钥交换的目的是计算另一个值，即预主密钥（premaster secret）。这个值是组成主密钥的来源。

TLS支持许多密钥交换算法，能够支持各种证书类型、公钥算法和密钥生成协议。它们之中有一些在TLS协议主规格书中定义，但更多的则是在其他规格说明中定义。你可以在表2-1中找到最常用的算法。

表2-1　最常用的密钥交换算法概览

密钥交换	描　述
dh_anon	Diffie-Hellman（DH）密钥交换，未经身份验证
dhe_rsa	临时DH密钥交换，使用RSA身份验证
ecdh_anon	临时椭圆曲线DH（elliptic curve DH，ECDH）密钥交换，未经身份验证（RFC 4492）
ecdhe_rsa	临时ECDH密钥交换，使用RSA身份验证（RFC 4492）
ecdhe_ecdsa	临时ECDH密钥交换，使用ECDSA身份验证（RFC 4492）
krb5	Kerberos密钥交换（RFC 2712）
rsa	RSA密钥交换和身份验证
psk	预共享密钥（pre-shared key，PSK）密钥交换和身份验证（RFC 4279）
dhe_psk	临时DH密钥交换，使用PSK身份验证（RFC 4279）
rsa_psk	PSK密钥交换，使用RSA身份验证（RFC 4279）
srp	安全远程密码（secure remote password，SRP）密钥交换和身份验证（RFC 5054）

使用哪一种密钥交换由协商的套件所决定。一旦套件决定下来，两端都能了解按照哪种算法继续操作。实际使用的密钥交换算法主要有以下4种。

- RSA

  RSA是一种事实上的标准密钥交换算法，它得到了广泛的支持。但它受到一个问题的严重威胁：它的设计使被动攻击者可以解码所有加密数据，只要她能够访问服务器的私钥。因此，RSA密钥交换正慢慢被其他支持前向保密（forward secrecy）的算法所替代。RSA密钥交换是一种密钥传输（key transport）算法，这种算法由客户端生成预主密钥，并以服务器公钥加密传送给服务器。

- DHE_RSA

  临时Diffie-Hellman（ephemeral Diffie-Hellman，DHE）密钥交换是一种构造完备的算法。它的优点是支持前向保密，缺点是执行缓慢。DHE是一种密钥协定算法，进行协商的团体都对密钥生成产生作用，并对公共密钥达成一致。在TLS中，DHE通常与RSA身份验证联合使用。

- ECDHE_RSA和ECDHE_ECDSA

  临时椭圆曲线Diffie-Hellman（ephemeral elliptic curve Diffie-Hellman，ECDHE）密钥交换

建立在椭圆曲线加密的基础之上。椭圆曲线算法是相对较新的算法。大家认可它执行很快而且提供了前向保密。但是只有较新的客户端才能较好地支持。ECDHE也是一种密钥协定算法，其理论原理与DHE类似。在TLS中，ECDHE可以与RSA或者ECDSA身份验证一起使用。

不论使用哪一种密钥交换，服务器都有机会发送ServerKeyExchange消息率先发话：

```
struct {
    select (KeyExchangeAlgorithm) {
        case dh_anon:
            ServerDHParams    params;
        case dhe_rsa:
            ServerDHParams    params;
            Signature         params_signature;
        case ecdh_anon:
            ServerECDHParams  params;
        case ecdhe_rsa:
        case ecdhe_ecdsa:
            ServerECDHParams  params;
            Signature         params_signature;
        case rsa:
        case dh_rsa:
            /* 无消息 */
    };
} ServerKeyExchange;
```

你可以在上面的消息定义中发现，在某些算法内，服务器不发送任何信息。原因是在这些情况下，所有需要的信息已经通过其他消息得到；不然，服务器就会在此发送其密钥交换的参数。关键的是，服务器也会发送参数的签名用于身份验证。使用签名，客户端得以确认它正在与持有私钥对应证书中的公钥的团体进行通信。

客户端会发送ClientKeyExchange消息传送它的密钥交换参数，这个消息总是必需的：

```
struct {
    select (KeyExchangeAlgorithm) {
        case rsa:
            EncryptedPreMasterSecret;
        case dhe_dss:
        case dhe_rsa:
        case dh_dss:
        case dh_rsa:
        case dh_anon:
            ClientDiffieHellmanPublic;
        case ecdhe:
            ClientECDiffieHellmanPublic;
    } exchange_keys;
} ClientKeyExchange;
```

2.3.1　RSA 密钥交换

RSA密钥交换的过程十分直截了当。客户端生成预主密钥（46字节随机数），使用服务器公

钥对其加密，将其包含在ClientKeyExchange消息中，最后发送出去。服务器只需要解密这条消息就能取出预主密钥。TLS使用的是RFC 3447[①]定义的RSAES-PKCS1-v1_5加密方案。

> **注意**
> 因为RSA算法可以同时用于加密和数字签名，所以RSA密钥交换可以按照这种方式工作。其他流行的密钥类型，比如DSA（DSS）和ECDSA，只能用于签名。

RSA密钥交换的简单性也是它最大的弱点。用于加密预主密钥的服务器公钥，一般会保持多年不变。任何能够接触到对应私钥的人都可以恢复预主密钥，并构建相同的主密钥，从而危害到会话安全性。

对目标的攻击并不需要实时进行，强大的对手可以制定长期行动。攻击者会记录所有加密的流量，耐心等待有朝一日可以得到密钥。比如，计算机能力的进步使暴力破解成为可能；也可以通过法律强制力、政治高压、贿赂或强行进入使用该密钥的服务器来取得密钥。只要密钥泄露，就可以解密之前记录的所有流量了。

TLS中其他常见的密钥交换方式都不受这个问题的影响，被称为支持前向保密。使用那些密钥交换时，每个连接使用的主密钥相互独立。泄露的服务器密钥可以用于冒充服务器，但不能用于追溯解密任何流量。

2.3.2 Diffie-Hellman 密钥交换

Diffie-Hellman（DH）密钥交换是一种密钥协定的协议，它使两个团体在不安全的信道上生成共享密钥成为可能[②]。

> **注意**
> 以这种方式协商共享密钥时不会受到被动攻击的威胁，但主动攻击者却可以劫持通信信道，冒充对端。这就是DH密钥交换通常与身份验证联合使用的原因。

抛开算法的细节，DH的诀窍是使用了一种正向计算简单、逆向计算困难的数学函数，即使交换中某些因子已被知晓，情况也是一样。最恰当的类比示例是混色：如果有两种颜色，那么很容易将其混在一起得到第三种颜色；但是如果只有第三种颜色的话，就很难确定究竟它是由哪两种颜色混合而成的[③]。

DH密钥交换需要6个参数：其中两个（dh_p和dh_g）称为域参数，由服务器选取。协商过程中，客户端和服务器各自生成另外两个参数，相互发送其中一个参数（dh_Ys和dh_Yc）到对端，

---

① RFC 3447: RSA Cryptography Specifications Version 2.1，http://tools.ietf.org/html/rfc3447（Jonsson和Kaliski，2003年2月）。

② Diffie-Hellman key exchange，https://en.wikipedia.org/wiki/Diffie–Hellman_key_exchange（维基百科，检索于2014年6月18日）。

③ Public Key Cryptography: Diffie-Hellman Key Exchange，https://www.youtube.com/watch?v=3QnD2c4Xovk（YouTube，检索于2014年6月26日）。

再经过计算，最终得到共享密钥。

临时Diffie-Hellman（ephemeral Diffie-Hellman，DHE）密钥交换中没有任何参数被重复使用。与之相对，在一些DH密钥交换方式中，某些参数是静态的，并被嵌入到服务器和客户端的证书中。这样的话，密钥交换的结果是一直不变的共享密钥，就无法具备前向保密的能力。

TLS支持静态DH密钥交换，但无人使用。在协商DHE套件时，服务器将其所有参数填入ServerDHParams块并发送：

```
struct {
    opaque dh_p;
    opaque dh_g;
    opaque dh_Ys;
} ServerDHParams;
```

客户端响应并发送其公开参数（dh_Yc）：

```
struct {
    select (PublicValueEncoding) {
        case implicit:
            /* 空的，当客户端公共参数嵌入其客户端时 */
        case explicit:
            opaque dh_Yc;
    } dh_public;
} ClientDiffieHellmanPublic;
```

当前使用的DH交换存在以下这些现实问题。

- **DH参数的安全性**

  DH密钥交换的安全性取决于域参数的质量。服务器发送弱的或者不安全的参数，将对会话的安全性造成损害。这个问题在一篇研究论文“Triple Handshake Attack”中进行过重点论述，弱DH参数被用作一种攻击向量[①]。

- **DH参数协商**

  TLS并没有为客户端提供传递期望使用的DH参数的强度的设施。比如，客户端可能希望避免使用弱参数，抑或可能不支持强参数。因此，选择DHE套件的服务器事实上只能期待DH参数可以被客户端接受。

- **参数强度不够**

  以历史角度来说，DH参数很大程度上被忽略了，其安全性也被忽视了。许多库和服务器默认使用弱DH参数，而且经常不提供配置DH参数强度的方法。因此，服务器使用1024位弱参数、768位非安全参数，更有甚者使用512位参数，这些情况都很常见。直到最近，一些平台才开始使用2048位或者更高位数的强参数。

  2015年5月披露的Logjam攻击表明，512位的DH参数在使用合适资源的情况下可以被攻击者在很短的时间内成功利用，同时可以估计厉害的攻击者甚至可能利用768位的参数。同样的研究还强调非常厉害的攻击者甚至有可能可以攻破那些被广泛使用的、长度为1024

① 要想了解有关三次握手攻击的更多信息，请参考7.6节。

位的标准参数组，从而以被动方式入侵数以百万计的互联网服务器。在6.5节中可以找到有关此问题的更多信息。

这些问题可以通过对不同强度的域参数定义来进行标准化，或者扩展TLS允许客户端告知其偏好的方法来解决[①]。

2.3.3 椭圆曲线 Diffie-Hellman 密钥交换

临时椭圆曲线Diffie-Hellman（elliptic curve Diffie-Hellman，ECDH）密钥交换原理与DH相似，但它的核心使用了不同的数学基础。正如名称所示，ECDHE基于椭圆曲线（elliptic curve，EC）加密。

ECDH密钥交换发生在一条由服务器定义的特定的椭圆曲线上。这条曲线代替了DH中域参数的角色。理论上，ECDH支持静态的密钥交换，但实际使用时，只使用了这种临时的变种（ECDHE）。

密钥交换由服务器发起，它选择一条椭圆曲线和公开参数（EC point）并提交：

```
struct {
    ECParameters curve_params;
    ECPoint public;
} ServerECDHParams;
```

服务器可以为密钥交换明确指定任意一条曲线，但TLS并未使用这个功能。作为替代，在TLS中，服务器通过指定某个名称引用一条可能预先定义好参数的曲线（命名曲线，named curve）：

```
struct {
    ECCurveType curve_type;
    select (curve_type) {
        case explicit_prime:
            /* 为了清晰，略去 */
        case explicit_char2:
            /* 为了清晰，略去 */
        case named_curve:
            NamedCurve namedcurve;
    };
} ECParameters;
```

然后客户端提交自己的公开参数。在那以后，就可以计算预主密钥：

```
struct {
    select (PublicValueEncoding) {
        case implicit:
            /* 空的 */
        case explicit:
            ECPoint ecdh_Yc;
    } ecdh_public;
} ClientECDiffieHellmanPublic;
```

使用预定义参数，以及`ellipic_curve`扩展（客户端可以提交支持的曲线），可以使服务器选

① Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for TLS，https://datatracker.ietf.org/doc/draft-ietf-tls-negotiated-ff-dhe/（D. Gillmor，2015年6月）。

择一条双方都支持的曲线。你可以在2.12.3节中找到更多有关命名曲线的可用信息。

2.4 身份验证

在TLS中，为了避免重复执行密码操作造成巨大开销，身份验证与密钥交换紧紧捆绑在一起。大多数场景中，身份验证的基础是证书支持的公钥密码（最常见的是RSA，有时也用ECDSA）。一旦证书验证通过，客户端就知道了使用的公钥。在此之后，客户端将公钥交给指定的密钥交换算法，并由它负责以某种方式使用公钥验证另一端。

在RSA密钥交换的过程中，客户端生成一个随机值作为预主密钥，并以服务器公钥加密后发送出去。拥有对应私钥的服务器解码消息得到预主密钥。身份验证原理很清楚：只有拥有对应私钥的服务器才能取得预主密钥，构造正确的会话密钥，并生成正确的`Finished`消息。

在DHE和ECDHE的交换过程中，服务器为密钥交换提供自己的参数，并使用自己的私钥签名。客户端持有对应的公钥（从已验证的证书中获得），可以验证参数是否真正出自期望的服务器。

> **注意**
> 服务器参数是与客户端和服务器随机值连在一起进行签名的，而客户端和服务器随机值对于握手来说是唯一的。因而，即使签名是以明文方式发送的，它也只对当前握手有效，这意味着攻击者无法重用该签名。Logjam攻击显示了这种将签名绑定在握手过程上面的弊端；主动网络攻击者可以同步产生这个随机值，并且在某些情况下再次利用服务器签名。

2.5 加密

TLS可以使用各种方法加密数据，比如使用3DES、AES、ARIA、CAMELLIA、RC4或者SEED等算法。目前使用最为广泛的加密算法是AES。TLS支持三种加密类型：序列密码、分组密码和已验证的加密。在TLS中，完整性验证是加密处理的一部分；它要么在协议级中显式处理，要么由协商的密码隐式处理。

2.5.1 序列加密

使用序列密码时，加密由两步组成。第一步，计算MAC值，范围包含记录序列号、标头、明文。MAC包含标头能确保未进行加密的标头不会遭受篡改。MAC包括序列号，能确保消息不被重放。第二步，加密明文和MAC，生成密文。整个过程如图2-5所示。

> **注意**
> 使用完整性验证但不进行加密的套件与使用序列密码加密的套件以相同的方式实现。明文被简单地复制到TLS记录中，而MAC则以这里描述的方法计算。

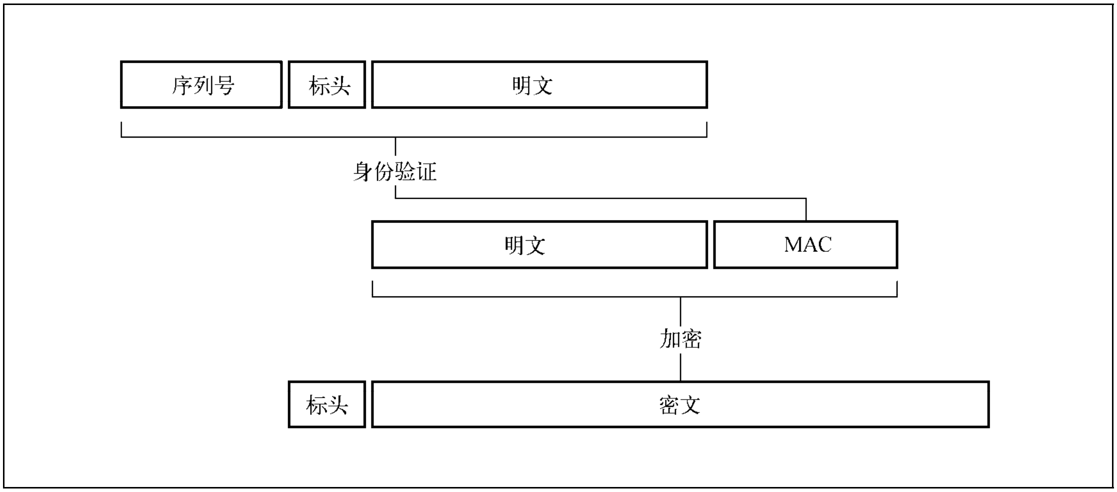


图2-5　序列加密

2.5.2　分组加密

使用分组密码时，加密会涉及更多内容，因为需要为分组加密的特性准备解决方案。具体来说，需要以下几个步骤，如图2-6所示。

(1) 计算序列号、标头和明文的MAC。

(2) 构造填充，确认加密前的数据长度是分组大小（通常16字节）的整数倍。

(3) 生成一个长度与分组大小一致的不可预期的*初始向量*（initialization vector，IV）。IV能保证加密是不确定的。

(4) 使用CBC分组模式加密明文、MAC和填充。

(5) 将IV和密文一起发送。

> **注意**
> 可以在1.4.1节找到有关CBC分组模式、填充和初始向量的更多信息。

这种处理方式被称为先计算MAC，*再加密*（MAC-then-encrypt），而它也是很多问题的源头。在TLS 1.1和更新的版本中，每条记录中都包含显式IV；而TLS 1.0和以往的版本则使用隐式IV（使用前一个TLS记录中的加密块作为下一块的IV），但这在2011年被发现并不安全[①]。

另一个问题是MAC计算不包括填充，这给主动网络攻击者进行*填充预示攻击*（padding oracle attack）提供了机会（有成功攻击TLS的示例[②]）。这里的问题其实是，协议定义的分组加密方式在现实中很难安全地实现。就我们所知，现在的实现并没有明显表现出易受攻击，但仍然不能对这

① 这个问题首先被称为BEAST的攻击所利用。关于BEAST，我将在7.2节中进行讨论。

② 我将在7.4节中讨论填充预示攻击。

个弱点掉以轻心。

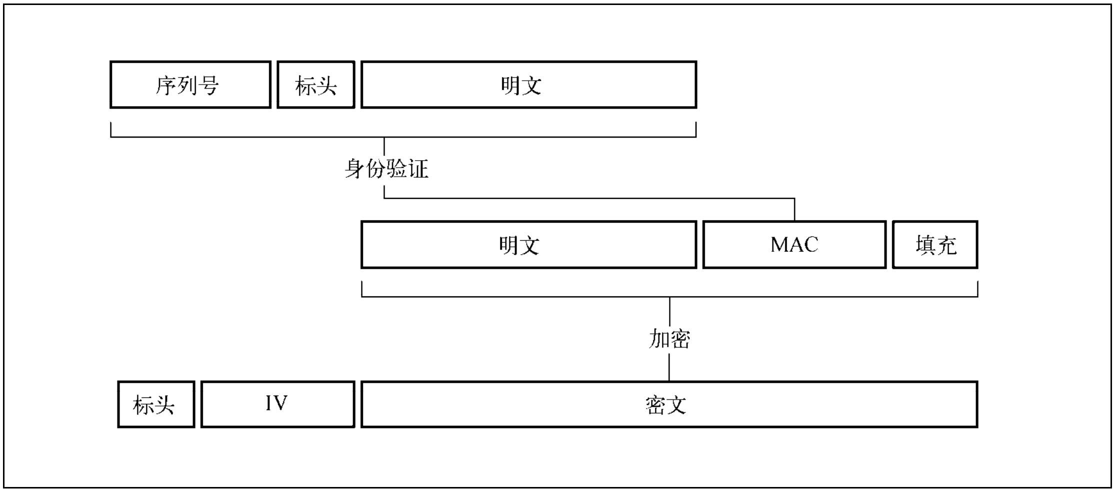


图2-6　分组加密

另一种处理安排方式的提案称为先加密，再计算MAC（encrypt-then-MAC），最近才被公开提出[①]。在这种替代方案中，首先对明文和填充进行加密，再将结果交给MAC算法。这可以保证主动网络攻击者不能操纵任何加密数据。

2.5.3　已验证的加密

已验证的密码将加密和完整性验证合二为一，全名是使用关联数据的已验证加密（authenticated encryption with associated data，AEAD）。表面上，它看起来是序列密码和分组密码的交叉。它不用填充[②]，也不用初始向量，而是使用一个特殊的值，称为nonce（在加密通信中仅使用一次的密钥）。这个值必须唯一。加密过程比使用分组密码要简单一些，如图2-7所示。

(1) 生成一个唯一的64位nonce。

(2) 使用已验证加密算法加密明文；同时也将序列号和记录标头作为完整性验证依据的额外数据交给算法。

(3) 将nonce和密文一起发送。

已验证加密被认为是当前TLS中可用的加密模式中最好的一种，因为它可以避免MAC-then-encrypt方式带来的问题。虽然TLS当前定义基于GCM和CCM块模式的已验证套件，实际上仅支持GCM套件。基于ChaCha20流密码的全新已验证套件当前正在进行标准化[③]。

---

① RFC 7366: Encrypt-then-MAC for TLS and DTLS，https://datatracker.ietf.org/doc/rfc7366/（Peter Gutmann，2014年9月）。

② 实际上，它们不一定会使用填充。即使它们使用的话，也是实现细节，不会暴露给TLS协议。

③ The ChaCha20-Poly1305 AEAD Cipher for TLS，https://datatracker.ietf.org/doc/draft-ietf-tls-chacha20-poly1305/（Langley等，2015年6月）。

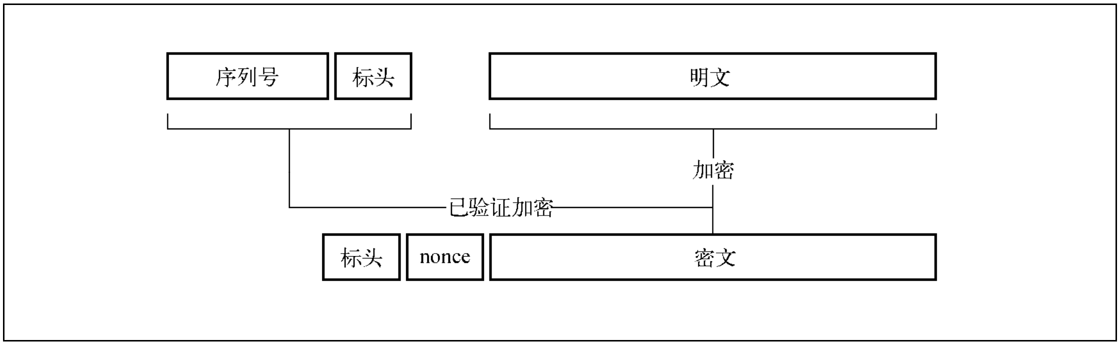


图2-7　已验证加密

2.6　重新协商

大部分TLS连接都以握手作为起点，经过应用数据的交换，最后关闭会话。但如果请求重新协商，就会发起一次新的握手，对新的连接安全参数达成一致。这个功能在以下这些情形下很有用。

- 客户端证书

  客户端证书并不常用，但因为它可以提供双因素身份验证，所以还是有一些网站在使用它。部署客户端证书有两种方法。你可以要求连接到网站的所有连接都需要客户端证书，但是这种方式对那些（还）没有证书的使用者并不友好；在连接成功以前，你无法向它们发送任何信息和说明。处理错误的情况同样不可能。因此，许多操作员会选择允许连接到网站根路径的连接不携带证书，并设计一个需要提供客户端证书的子区域。当用户打算浏览子区域时，服务器发起重新协商请求，要求客户端提供证书。

- 隐藏消息

  上述的双步控制方式使客户端证书具备了一个额外的优势：因为第二次握手是加密的，这意味着被动攻击者无法监视协商，而且最关键的是攻击者根本无法观察到客户端证书。这解决了非常重要的潜在隐私问题，因为客户端证书通常包含身份识别信息。比如，Tor协议就是用这种方式进行重新协商的①。

- 改变加密强度

  以前，当网站加密刚刚出现（而且是CPU密集的）的时候，经常可以发现网站将加密配置分成两个级别。你可以默认使用比较弱的加密，而在特定区域使用强加密②。如果要使用客户端证书，则通过重新协商实现这个功能。当你尝试进入网站中更安全的子区域时，服务器将请求更强的安全性。

① Tor Protocol Specification，https://gitweb.torproject.org/torspec.git?a=blob_plain;hb=HEAD;f=tor-spec.txt（Dingledine和Mathewson，检索于2014年6月30日）。

② 这种想法本身就是问题。你的加密要么足够安全要么就是不安全。如果你的敌对方可以攻入较弱的配置，他们就可以完全控制牺牲者的浏览器。做到那样的话，他们可以欺骗牺牲者将所有秘密揭示出来（比如密码）。

此外，在以下两种情况下协议需要使用重新协商，不过这两种情况在实际工作中都不大可能发生。

- 服务网关密码

  20世纪90年代，美国还未允许出口强密码算法，一个称为服务网关密码（Server-Gated Crypto，SGC）的功能使美国供应商能够将强密码算法输出到世界范围，但仅对选定的（主要是金融业的）美国网站有效。浏览器默认使用的是弱加密，遇到特定证书以后升级至强证书。整个升级过程由客户端控制，通过重新协商实现。只有少数选定的CA获得允许签发这种特定证书。密码输出限制于2000年解除，SGC随之废弃。

- TLS记录的计数器溢出

  TLS内部将数据包装成记录，并为每个记录指定唯一的64位序列号。每当发生记录交换时，序列号就随之增长。一旦序列号接近溢出，协议就会强制执行重新协商。然而，因为这个计数器本身的数字就非常大，所以实践中不太可能出现溢出。

协议允许客户端在任意时间简单地发送新的ClientHello消息请求重新协商，就如同建立一个全新的连接一样，这被称为客户端发起的重新协商（client-initiated renegotiation）。

如果服务器希望重新协商，它会发送HelloRequest协议消息给客户端。这个消息通知客户端停止发送应用数据，并开始新的握手，这被称为服务器发起的重新协商（server-initiated renegotiation）。

正如原本设计的那样，重新协商并不安全，并且可被主动网络攻击者以很多方式滥用。它的弱点于2009年被发现[①]，然后通过引进renegotiation_info扩展得以修正。我将在本章后面对这个扩展进行讨论。

2.7　应用数据协议

应用数据协议携带着应用消息，只以TLS的角度考虑的话，这些就是数据缓冲区。记录层使用当前连接安全参数对这些消息进行打包、碎片整理和加密。

2.8　警报协议

警报的目的是以简单的通知机制告知对端通信出现异常状况。它通常会携带close_notify异常，在连接关闭时使用，报告错误。警报非常简单，只有两个字段：

```
struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

AlertLevel字段表示警报的严重程度，可取值warning或者fatal。AlertDescription直接表示警报代码。不论这种设计是好是坏，警报都没有表达任意信息的能力，比如实际的错误提示。

① 要了解更多信息，请参考7.1节。

严重程度为fatal的消息会立即终止当前连接并使会话失效（相同会话的其他正在进行的连接会继续，但会话绝不可能恢复了）。发送警告通知的一端不会主动终止连接，而是交由接收端通过发送它自己的严重警报对该警告自行作出反应。

2.9　关闭连接

关闭连接警报（closure alert）用于以有序的方式关闭TLS连接。一旦一端决定关闭连接，就会发送一个close_notify警报。另一端收到这个警报以后，会丢弃任何还未写出的数据，并发送自己的close_notify警报。在警报之后到来的任何消息都将被忽略。

这个关闭协议虽然简单，但它可以避免截断攻击，也就是主动攻击者打断通信过程，阻断所有后续消息的攻击，因而是必需的。如果没有关闭协议，通信双方就无法确认是遭到攻击还是通信真正结束。

> **注意**
> 虽然协议自身不易遭到截断攻击，但是其许多实现却容易遭到攻击，因为连接关闭协议的冲突非常普遍。我将在6.7节中讨论这个问题。

2.10　密码操作

本节会对协议中一些重要的方面进行简单讨论：伪随机函数、构建主密钥和生成连接密钥。

2.10.1　伪随机函数

在TLS中，伪随机函数（pseudorandom function，PRF）用于生成任意数量的伪随机数据。PRF使用一条秘密、一颗种子和一个唯一标签。从TLS 1.2起，所有的算法套件都需要明确指定它们的PRF。所有TLS 1.2套件都使用基于HMAC和SHA256的PRF。以TLS 1.2协商的过往老套件也会使用与此相同的PRF。

TLS 1.2定义的PRF基于数据扩展函数P_hash，这个函数使用了HMAC和一个任意散列函数：

```
P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                       HMAC_hash(secret, A(2) + seed) +
                       HMAC_hash(secret, A(3) + seed) + ...
```

A(i)函数定义如下：

```
A(1) = HMAC_hash(secret, seed)
A(2) = HMAC_hash(secret, A(1))
...
A(i) = HMAC_hash(secret, A(i-1))
```

PRF则是结合标签和种子对P_hash的封装：

```
PRF(secret, label, seed) = P_hash(secret, label + seed)
```

引入种子和标签允许在不同的环境中重用相同的秘密，而且能够生成不同的输出（因为标签和种子不同）。

2.10.2 主密钥

前面大家已经看到，密钥交换过程的输出是预主密钥。对这个值进行进一步加工，就是使用PRF生成48字节（384位）主密钥：

```
master_secret = PRF(pre_master_secret, "master secret",
                    ClientHello.random + ServerHello.random)
```

因为使用不同的密钥交换方法，得到的预主密钥长度可能不同，所以需要执行这个步骤。同时，因为客户端和服务器的随机字段被用作种子，所以主密钥实际上也是随机的[①]，且与协商握手绑定。

> **注意**
> 主密钥和握手之间的结合点只是计算主密钥依赖交换的随机数，并已经表现出这是不充分的。攻击者可以观察并复制这些值，从而创建共用相同主密钥的多个会话。这个弱点已被之前提到的三次握手攻击所利用[②]。

2.10.3 密钥生成

连接所需的密钥材料是用单一的PRF调用基于主密钥和客户端、服务器的随机数生成的：

```
key_block = PRF(master_secret, "key expansion",
                server_random + client_random)
```

密钥块的长度根据协商的参数而有所不同。密钥块分为六个密钥：两个MAC密钥、两个加密密钥和两个初始向量（只在必要时生成；序列密码不会使用IV）。AEAD套件不使用MAC密钥。不同的密钥用于不同的操作，这样可以预防当共享相同密钥时，密钥学基元之间出现不可预见的交互。同样，因为客户端和服务器都拥有各自的一组密钥，由其中一方产生的消息不会被解释成是由另一方产生的。这个设计决策使协议更加可靠。

> **注意**
> 当恢复会话时，在生成密钥块时使用相同的主密钥，但PRF以当前握手时客户端和服务器的随机值进行种子设定。因为每次握手时的随机值都不同，所以密钥每次也不同。

① 虽然绝大多数平时常用的密钥交换机制每次都会生成不同的预主密钥，但也有一些机制依赖长期密钥，因此会重用相同的预主密钥。所以必须使用随机化以确保密钥不会重复。

② 要想了解有关三次握手攻击的更多信息，请参考7.6节。

2.11 密码套件

如你所见，TLS为实现所需的安全属性提供了非常大的灵活性。它是一个创造实际密码协议的框架。虽然以往版本将某些加密基元硬编码到了协议中，但TLS 1.2是完全可配置的。密码套件是一组选定的加密基元和其他参数，它可以精确定义如何实现安全。套件大致由以下这些属性定义。

- 身份验证方法
- 密钥交换方法
- 加密算法
- 加密蜜钥大小
- 密码模式（可应用时）
- MAC算法（可应用时）
- PRF（只有TLS 1.2一定使用，其他版本取决于各自协议）
- 用于Finished消息的散列函数（TLS 1.2）
- verify_data结构的长度（TLS 1.2）

密码套件都倾向于使用较长的描述性名称，并且相当一致：它们都由密钥交换方法、身份验证方法、密码定义以及可选的MAC或PRF算法组合而成[①]，如图2-8所示。



图2-8 密码套件名称构成

虽然套件名称传达所有安全性参数时并不充分，但借其可以轻松推断出最重要的那些参数。此外，其他参数则可以在RFC上通过套件定义找到。你可以在表2-2中看到少量精选套件的安全性属性。在撰写这本书的时候，已经有超过300种官方密码套件，所以不能全部罗列在这里。如果想得到完整列表，请访问IANA上的TLS官方页面[②]。

表2-2 密码套件的名称和其安全性属性的示例

密码套件名称	身份验证	密钥交换	密码	MAC	PRF
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	RSA	ECDHE	AES-128-GCM	—	SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	ECDSA	ECDHE	AES-256-GCM	—	SHA384

① TLS套件会使用TLS_前缀，而SSL 3套件使用SSL_前缀，SSL 2套件使用SSL_CK_前缀。在所有场景中，这个命名方法大致都是相同的。可是，并非所有供应商都使用标准套件命名。OpenSSL和GnuTLS就使用不同的命名。Microsoft基本上会使用标准命名，但有时使用前缀扩展它们，以指明ECDHE密钥交换的强度。

② TLS Parameters，https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml（IANA，检索于2014年6月30日）。

（续）

密码套件名称	身份验证	密钥交换	密码	MAC	PRF
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	RSA	DHE	3DES-EDE-CBC	SHA1	协议
TLS_RSA_WITH_AES_128_CBC_SHA	RSA	RSA	AES-128-CBC	SHA1	协议
TLS_ECDHE_ECDSA_WITH_AES_128_CCM	ECDSA	ECDHE	AES-128-CCM	—	SHA256

随着TLS 1.2的引进（允许定义额外定制的参数，比如PRF）以及已验证套件的引进，为了全面解析密码套件名称，需要在某种程度上理解其实现。

- 已验证套件结合了密码学的身份验证和加密，这意味TLS这一层无需执行完整性验证。GCM套件使用名称的最后一段指明使用的PRF算法，而非MAC算法。另外CCM套件则将最后一段完全省略。
- TLS 1.2是唯一允许套件定义其PRF的协议。对于TLS 1.2以前就已定义的套件，由协商的协议版本控制PRF。以TLS_RSA_WITH_AES_128_CBC_SHA套件为例，当协商后的协议是TLS 1.2时，它使用的PRF是基于HMAC-SHA256；当协议是TLS 1.0时，PRF则是基于HMAC-MD5/HMAC-SHA1。另一方面，SHA384 GCM套件（只在TLS 1.2及更高版本中可用）则会一直使用HMAC-SHA384作为PRF。

> **注意**
> 密码套件名称使用简化符号表示MAC算法，只指定其散列函数。这经常会在散列函数存在弱点时引发大家的迷惑。比方说，虽然已知SHA1在选择前缀攻击面前是弱点，但它在TLS中却并不存在此弱点，因为它是用在HMAC体系中。目前尚无值得注意的针对HMAC-SHA1的攻击。

密码套件并未完全掌控其安全参数。它们只是定义了最关键的身份验证和密钥交换算法，而对这些算法的实际参数并没有控制能力（比如密钥和参数强度）。

> **注意**
> 密码套件只能与其预期的特定身份验证机制一起使用。比方说，名称含有ECDSA的套件需要ECDSA密钥。如果一台服务器中只有一个RSA密钥，那么它会展示自己不支持任何ECDSA套件。

对于身份验证，其强度主要依靠证书，更确切地说是证书中的密钥长度和签名算法。RSA密钥交换的强度也依赖证书。可以为DHE和ECDHE密钥交换配置不同的强度，这通常是在服务器级别的配置中完成的。某些服务器将这些配置暴露给最终用户，而另一些不会。我会在第8章及其后的技术性章节中更详细地讨论这些方面。

2.12　扩展

TLS扩展是一种通用目的的扩展机制，使用这种机制可以在不修改协议本身的条件下为TLS

协议增加功能。它在2003年作为一个单独的规格说明（RFC 3456）首次出现，但随后即被加入到TLS 1.2中。

扩展以扩展块的形式加在ClientHello和ServerHello消息的末尾：

```
Extension extensions;
```

扩展块由所需数量的扩展一个个堆叠而成。每一个扩展标头是2字节扩展类型（唯一标志），后接扩展数据：

```
struct {
    ExtensionType extension_type;
    opaque extension_data;
} Extension;
```

扩展的格式和期望的行为由每个扩展自己决定。在实践中，扩展通常用于通知支持某些新功能（因此改变了协议），以及用于在握手阶段传递所需的额外数据。自从扩展被引入TLS，它就成为了TLS演进的主要载体。

在本节中，我将讨论最常见的TLS扩展（如表2-3所示）。因为IANA一直保持对扩展类型的跟踪，所以可以从其网站上获得官方的扩展列表①。

表2-3 常见的TLS扩展

类 型	名 称	描 述
0	server_name	包含连接欲访问的安全虚拟主机
5	status_request	指示支持OCSP stapling
13 (0x0d)	signature_algorithms	包含支持的签名算法/散列函数对
15 (0x0f)	heartbeat	指示支持心跳协议
16 (0x10)	application_layer_protocol_negotiation	包含客户端希望协商的并且支持的应用层协议
18 (0x12)	signed_certificate_timestamp	服务器用来提交证据，以证明证书已被公众共享；是证书透明度的一部分
21 (0x15)	padding	用于解决F5负载均衡设备[a]中的特定bug
35 (0x23)	session_ticket	指示支持无状态会话恢复
13172 (0x3374)	next_protocol_negotiation	指示支持次协议协商
65281 (0xff01)	renegotiation_info	指示支持安全重新协商

a A TLS padding extension，https://datatracker.ietf.org/doc/draft-ietf-tls-padding/（Internet-Draft，A. Langley，2015年2月）。

2.12.1 应用层协议协商

应用层协议协商（application layer protocol negotiation，ALPN）协议扩展能够在TLS连接上协商不同的应用层协议②。使用ALPN，一个监听443端口的服务器可以默认提供HTTP 1.1，并允

① TLS Extensions，https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml（IANA，检索于2014年6月30日）。

② RFC 7301: TLS Application-Layer Protocol Negotiation Extension，http://tools.ietf.org/html/rfc7301（Friedl等，2014年7月）。

许协商其他协议，比如SPDY或者HTTP 2.0。

支持ALPN的客户端在application_layer_protocol_negotiation扩展中提交自己支持的应用层协议列表给服务器。兼容的服务器会决定使用的协议并使用相同扩展向客户端通知其决定。

ALPN与其老前辈NPN（会在本节中稍后进行讨论）提供的主要功能一致，只是在次要属性上有所不同。NPN偏爱将协议选择结果加密，而ALPN则是以明文传输，使中间设备可以检查它们并根据观察所得信息为流量选择路由。

2.12.2　证书透明度

证书透明度（certificate transparency）[①]还是一个提案，目的是通过保持所有公开的服务器证书来改进互联网PKI。它的基本想法是CA将每一张证书都提交给一组公开的日志服务器，反过来，这些CA将收到提交的证明，称为*已签名证书时间戳*（signed certificate timestamp，SCT），并中继给最终用户。有一些选项用来传送SCT，其中之一就是新的TLS扩展signed_certificate_timestamp。

2.12.3　椭圆曲线功能

RFC 4492提出了两个扩展，可以在握手时通告客户端的EC功能。elliptic_curves扩展在ClientHello中列出支持的曲线名称，使服务器可以在其中选择一条双方都支持的曲线。

```
struct {
    NamedCurve elliptic_curve_list
} EllipticCurveList;
```

主要的曲线在RFC 4492[②]中说明。所有的曲线都在标准主体（比如NIST[③]）之上定义，通过参数来实现：

```
enum {
    sect163k1 (1), sect163r1 (2), sect163r2 (3),
    sect193r1 (4), sect193r2 (5), sect233k1 (6),
    sect233r1 (7), sect239k1 (8), sect283k1 (9),
    sect283r1 (10), sect409k1 (11), sect409r1 (12),
    sect571k1 (13), sect571r1 (14), secp160k1 (15),
    secp160r1 (16), secp160r2 (17), secp192k1 (18),
    secp192r1 (19), secp224k1 (20), secp224r1 (21),
    secp256k1 (22), secp256r1 (23), secp384r1 (24),
    secp521r1 (25),
    reserved (0xFE00..0xFEFF),
    arbitrary_explicit_prime_curves(0xFF01),
    arbitrary_explicit_char2_curves(0xFF02)
} NamedCurve;
```

① Certificate Transparency，http://www.certificate-transparency.org/（Google，检索于2014年6月30日）。

② RFC 4492: ECC Cipher Suites for TLS，http://tools.ietf.org/html/rfc4492（S. Blake-Wilson等，2006年5月）。

③ FIPS 186-3: Digital Signature Standard，http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf（NIST，2009年6月）。

Brainpool曲线在稍后的RFC 7027[1]中定义。另外两条曲线Curve25519和 Curve448，当前正在进行标准化，以在TLS中使用[2]。

现在，只有两种曲线得到了广泛支持：`secp256r1`和`secp384r1`。一般来讲，根本不可能支持任意曲线[3]。

> **NIST椭圆曲线**
>
> NIST椭圆曲线有时会受到怀疑，因为它们没有解释参数是如何选择出来的[4]。尤其是在Dual EC DRBG后门曝光以后，任何没有解释的曲线都会受到怀疑。大家担心那些命名曲线存在只有设计者才知道而不为大众所知的缺陷。因此，要扩展TLS支持，其他曲线还需要做很多工作。

另一个定义的扩展是`ec_point_formats`。这个扩展可以在协商时对椭圆曲线顶点进行可选压缩。理论上，使用压缩的顶点格式可以在受限的环境下节省宝贵的带宽资源，但实际使用中节省的量不大（比如为256位的曲线节约64字节）[5]，因此压缩格式一般未被使用。

2.12.4 心跳

心跳（Heartbeat）[6]是一个协议扩展，添加了支持连接保活的功能（检查对端是否仍然可用），以及为TLS和DTLS发现路径最大传输单元（path maximum transmission unit，PMTU）[7]。虽然TLS通常用于TCP协议之上，而且TCP本身已经具备连接保活的功能，但是心跳的目标定位是DTLS，因为其工作在不可依赖的协议（比如UDP）之上。

> **注意**
>
> 有些人建议使用0长度的TLS记录（协议允许）来实现连接保活功能。但在试图减轻BEAST攻击的实际过程中显示了大量应用无法容忍没有任何数据的记录。并且，因为PMTU需要各种长度的负载，所以0长度TLS记录对其没有任何帮助。

---

① RFC 7027: ECC Brainpool Curves for TLS，http://tools.ietf.org/html/rfc7027（J. Merkle和M. Lochter，2013年10月）。

② Curve25519 and Curve448 for TLS，https://datatracker.ietf.org/doc/draft-ietf-tls-curve25519/（S. Josefsson和M. Pegourie-Gonnard，2015年7月）。

③ 生成性质良好且不限数量的椭圆曲线是一项复杂并且很容易出错的工作，大多数开发人员都会选择敬而远之。此外，命名曲线是可以进行优化以使之运行更快速的。

④ SafeCurves: choosing safe curves for elliptic-curve cryptography，http://safecurves.cr.yp.to/（D. J. Bernstein，检索于2014年6月30日）。

⑤ 这里想当然地未提及未压缩时需要的字节数，所以这个例子说明不了什么问题。——译者注

⑥ RFC 6520: TLS and DTLS Heartbeat Extension，https://tools.ietf.org/html/rfc6520（R. Seggelmann等，2012年2月）

⑦ 最大传输单元（maximum transmission unit，MTU）是能作为一个整体发送的数据最大长度。两端在直接通信时，它们可以交换其MTU。但是，当通信穿越许多跃点以后，有时需要渐进发送更大的数据帧来发现适用于整个路径的MTU。

首先，客户端和服务器通过心跳扩展相互通告支持心跳。在协商过程中，一方通过发送带有HeartbeatMode参数的心跳请求给另一方授权：

```
struct {
    HeartbeatMode mode;
} HeartbeatExtension;

enum {
    peer_allowed_to_send (1),
    peer_not_allowed_to_send (2)
} HeartbeatMode;
```

心跳是作为TLS的子协议实现的，这意味着心跳消息可能与应用数据甚至其他协议消息相互交错。按照RFC的说明，只允许在握手完成后才能发送心跳消息，但实际上，OpenSSL在TLS扩展交换完成以后就立即允许其发送。

现在还不清楚生产中有无使用心跳。但是，OpenSSL支持它并且默认启用。GnuTLS也实现了这个扩展。2014年4月以前，几乎没有人知道心跳是什么；在那之后，由于OpenSSL实现遭受针对其一个严重弱点的攻击，使得敏感数据从服务器进程的内存空间中外泄，人们才发现心跳的存在。那次利用这一漏洞的攻击被称为*心脏出血*（Heartbleed），也许是发生在TLS上的最严重的攻击。你可以在6.3节中阅读到更多相关信息。

2.12.5　次协议协商

当Google开始设计SPDY[①]（这个协议将比HTTP有所提升）时，它需要一种可靠的协议协商机制，可以与严格的防火墙和有问题的代理一起工作。因为SPDY本来就会使用TLS，所以它们决定为TLS提供一个协商应用层协议的扩展。其最终结果是*次协议协商*（next protocol negotiation，NPN）。

> **注意**
> 如果你调查NPN，可能会碰到许多不同版本的规格说明书。其中一些版本是TLS工作组在标准化讨论时生成的，而生产中使用的是一个更老的版本[②]。

启用SPDY支持的客户端提交的TLS握手中集成了一个空的next_protocol_negotiation扩展，但这仅在握手中也包含server_name扩展表明其需要访问的主机名时才行。兼容的服务器会返回含有next_protocol_negotiation扩展的响应，这次会包含服务器所支持的应用层协议的列表。

客户端通过一个新的握手消息NextProtocol表明期望的应用层信息：

```
struct {
    opaque selected_protocol;
    opaque padding;
```

① SPDY，https://en.wikipedia.org/wiki/SPDY（维基百科，检索于2014年6月30日）。

② Google Technical Note: TLS Next Protocol Negotiation Extension，https://web.archive.org/web/20150529234248/https://technotes.googlecode.com/git/nextprotoneg.html（Adam Langley，2012年5月）。

```
} NextProtocol;
```

为了避免被动攻击者得到客户端的选择，提交的消息被加密，这意味着客户端必须在ChangeCipherSpec消息以后才能发送它，而且还要在Finished消息之前。这就与标准握手的消息流有所不同。客户端不仅可以从服务器提供的列表中选择所期望的协议名称，而且也可以自由提交不在服务器通知中的协议。扩展会使用填充，这样可以隐藏其真实长度，这样敌对方无法通过观察加密消息的长度猜测选择的协议。

NPN已提交到TLS工作组，希望成为标准[1]。但即使其在业界得到广泛支持（比如Chrome、Firefox和OpenSSL），它仍然未被接受。引入新的握手消息，改变通常的握手流程，被认定是破坏性的并且过度复杂。大家也担心其不具备让中间设备了解协商的协议的能力，并且在实践中可能也存在问题。最终，工作组采用的是与之竞争的ALPN提案[2]。Google现在同时支持ALPN和NPN，而且将在未来切换到只支持ALPN[3]。

2.12.6 安全重新协商

renegotiation_info扩展以验证重新协商的双方仍是先前完成握手的两个团体方式来改进TLS。

开始（在某个连接的第一次握手期间），这个扩展用于双方相互通知对方自己支持安全重新协商；为了做到这一点，他们简单地发送不带数据的扩展。SSL 3不支持扩展，为了使其支持这种安全性，作为替代，客户端会发送特殊的通知套件TLS_EMPTY_RENEGOTIATION_INFO_SCSV（0xff）。

在后续的握手中，扩展用于提交先前握手的一些信息作为证明。客户端以先前的Finished消息作为verify_data的值发送出去。而服务器会发送两个值：首先是客户端的verify_data，接下来是服务器自己的。因为Finished消息总是加密的，所以攻击者得不到这些值。

2.12.7 服务器名称指示

服务器名称指示（server name indication，SNI）通过server_name扩展实现[4]，它可以为客户端提供一种机制，利用这种机制客户端可以告知服务器它希望与之建立连接的服务器的名称。换言之，这个扩展为安全虚拟主机提供支持：它为服务器提供足够的信息，使之可以在所有可用的安全虚拟主机中寻找到匹配的证书。如果没有这种机制，每个IP地址上只能部署一张证书[5]。因为SNI在TLS中添加得比较晚（2006），所以仍然存在很多陈旧的产品（比如Windows XP和一些

① Next Protocol Negotiation 03，http://www.ietf.org/mail-archive/web/tls/current/msg08678.html（Adam Langley，2012年4月24日）。

② Some missing context (was: Confirming consensus for ALPN)，http://www.ietf.org/mail-archive/web/tls/current/msg09344.html（Yoav Nir，2013年3月15日）。

③ NPN and ALPN，https://www.imperialviolet.org/2013/03/20/alpn.html（Adam Langley，2013年3月20日）。

④ RFC 6066: TLS Extensions: Extension Definitions，http://tools.ietf.org/html/rfc6066（D. Eastlake 3rd，2011年1月）。

⑤ 虽然HTTP也具有通过Host请求头发送主机信息的能力，但这是在应用层发送的，只能在TLS握手成功以后才能通告至服务器。

早期安卓版本）不支持这个扩展。因此，为大量受众提供开放站点建设服务的虚拟安全托管方案仍然不现实。

2.12.8　会话票证

会话票证（session ticket）引入了一种新的会话恢复机制，这种机制不需要任何服务器端存储[①]。其思想是服务器取出它的所有会话数据（状态）并进行加密，再以票证的方式发回客户端。在接下来的连接中，客户端将票证提交回服务器，由服务器检查票证的完整性，解密其内容，再使用其中的信息恢复会话。这种方法有可能使扩展服务器集群更为简单，因为如果不使用这种方式，就需要在服务集群的各个节点之间同步会话。

> **警告**
> 会话票证破坏了TLS安全模型。它使用票证密钥加密的会话状态并将其暴露在线路上。有些实现中的票证密钥可能会比连接使用的密码要弱。比如，OpenSSL使用128位的AES密钥。同时，相同的票证密钥在许多会话中重用，这就与RSA密钥交换的情形类似，无法提供前向保密；如果票证密钥被暴露，就可以解密连接上的全部数据。因此，使用会话票证时，票证密钥需要频繁轮换。

客户端可以使用一个空的会话票证扩展指示支持这种恢复方式。如果它希望恢复此前的会话，它应该将票证放置在扩展中代替空值。兼容服务器如希望发起一个新的票证，则应在其`ServerHello`中包含一个空`session_ticket`扩展。之后服务器便会等待客户端的`Finished`消息，验证它，将票证置入`NewSessionTicket`握手消息中发回客户端。如果服务器希望恢复此前的会话，它会按照简短握手的方式响应客户端，并且按照标准的会话恢复方式处理自身逻辑。

> **注意**
> 当服务器决定使用会话票证进行会话恢复时，它会发回一个空的会话ID字段（在其`ServerHello`消息中）。就这一点来说，这个会话不具有唯一的标识。然而，票证规格说明允许客户端在后续使用票证的握手中选择并提交一个会话ID（在其`ClientHello`中）。服务器如果接受票证，那么也必须以同样的会话ID作为响应。这就是为什么即使使用会话票证作为会话恢复机制，TLS Web服务器日志中也出现会话ID的原因。

2.12.9　签名算法

`signature_algoritms`扩展是在TLS 1.2中定义的。它使客户端可以通告自己支持的各种签名和散列算法。TLS规格说明书中列出了RSA、DSA和ECDSA签名算法，以及MD5、SHA1、SHA224、SHA256、SHA384和SHA512这些散列算法。使用`signature_algorithms`扩展，客户端可以提交其支持的签名–散列算法对。

---

① RFC 5077: TLS Session Resumption without Server-Side State，http://tools.ietf.org/html/rfc5077（Salowey等，2008年1月）。

这个扩展是可选的；如果未设置，服务器会根据客户端提供的密码套件推断其支持的签名算法（比如RSA套件可以推断出RSA签名，ECDSA套件支持ECDSA，等等），而且假定其支持SHA1。

2.12.10　OCSP stapling

客户端使用`status_request`扩展[①]指示支持OCSP stapling。服务器使用这个特性发送最新的证书吊销信息给客户端。（我会在5.10节对吊销进行详细讨论）。支持OCSP stapling的服务器在其`ServerHello`中返回一个空的`status_request`扩展，并在Certificate消息之后紧跟一条`CertificateStatus`握手消息，将OCSP响应（使用DER格式）包含在这条消息中。

OCSP stapling只支持一个OCSP响应，只能用于检测一张服务器证书的吊销状态。这个限制在RFC 6961[②]中得到了解决，因为它添加了对多个OCSP响应的支持（使用`status_request_v2`扩展来表明对它的支持）。但是，直到现在，这个改进版仍然没有得到客户端和服务器软件的完善支持。

2.13　协议限制

因为TLS在OSI层次中的定位和某些设计安排，除了那些偶然的弱点（那些弱点我将在后续章节中详细讨论）之外，当前它仍有几处众所周知的限制。

- 加密保护TCP连接的内容，但TCP和所有其他更低层的协议的元数据仍然是明文传输。因此，被动观察者可以确定源和目标的IP地址。这类信息的泄露不是TLS的责任，而是我们当前的分层网络模型导致的限制。
- 即使在TLS层，也有很多信息以明文形式暴露出去。第一次握手一定是非加密的，可以让被动观察者：(1) 了解客户端的功能，并使用其作为指纹；(2) 检查SNI信息确定期望访问的虚拟主机；(3) 检查主机证书，以及何时会使用客户端证书；(4) 存在得到足够信息以识别用户身份的可能性。这些问题有方法可以避免，但是这些方法都没有被主流实现所采用。
- 当启动加密以后，某些协议信息仍能被清楚地探查到：观察者可以了解到子协议和每条消息的长度。根据不同的协议，这些长度可以揭示某些底层通信的线索。比如，有一些研究尝试根据指明的请求和相应长度推断HTTP访问的是哪些资源。如果没有隐藏长度，不可能安全地在加密之前使用压缩（现在最常见的实践方法）。

网络层元数据的泄露只能在网络层解决。而其他限制是可以修复的，确实有一些解决问题的提案和讨论。你可以在本书后面的部分对这些问题有更多了解。

---

① RFC 6066: TLS Extensions: Extension Definitions，http://tools.ietf.org/html/rfc6066（D. Eastlake 3rd，2011年1月）。

② RFC 6961: TLS Multiple Certificate Status Request Extension，http://tools.ietf.org/html/rfc6961（Y. Pettersen，2013年6月）。

2.14 协议版本间的差异

本节描述SSL和TLS协议的各版本之间的主要差别。自从SSL 3以来，协议核心并没有大幅改变。TLS 1.0为了迎合使用另一个名称进行了有限的改变，发布TLS 1.1的首要目标是为了解决几个安全性问题。TLS 1.2引入了已验证加密，清理了散列，另外去掉了协议中的硬编码基元。

2.14.1 SSL 3

SSL 3于1995年年末发布。为了弥补先前协议版本的诸多弱点，SSL 3从头开始设计了一套协议，并一直沿用到了最新版本的TLS。如果你想更好地了解SSL 3作出了哪些改变以及作出改变的原因，我推荐Wagner和Schneier的协议分析论文①。

2.14.2 TLS 1.0

TLS 1.0于1999年1月发布。与SSL 3相比，它包含了以下改进。

- 这是定义基于标准HMAC的PRF的第一个版本。它将PRF以HMAC-MD5和HMAC-SHA的结合（XOR）实现。
- 生成主密钥使用PRF，而不是定制的构造方法。
- `verify_data`的值基于PRF，而不是定制的构造方法。
- 使用官方HMAC作为完整性验证（MAC）。SSL 3使用的是更早的、已被废弃的HMAC版本。
- 修改填充格式，使其更为可靠。2014年10月，被称为POODLE的攻击暴露了SSL 3的填充机制不安全。
- 去掉了FORTEZZA②套件。

协议清理的结果是TLS 1.0得到了FIPS的批准，允许其用于美国政府机构，这是现实中一个非常重要的事件。

如果你想研究TLS 1.0和之前版本的协议，我推荐Eric Rescorla的*SSL and TLS: Designing and Building Secure Systems*一书（Addison-Wesley，2001年出版）。我发现这本书对于理解TLS某些决定背后的理由，以及跟进其设计的演变，有着非常宝贵的价值。

2.14.3 TLS 1.1

TLS 1.1于2006年4月发布。与TLS 1.0相比，它包含以下主要改进。

- CBC加密使用包含在每个TLS记录中的显式IV。这弥补了IV可预测的弱点。不然这个弱点后面会被BEAST攻击所利用。
- 为了抵抗填充攻击，要求实现使用`bad_record_mac`警报作为填充问题的响应。不再赞成使

① Analysis of the SSL 3.0 protocol，https://www.schneier.com/paper-ssl-revised.pdf（David Wagner和Bruce Schneier，Proceedings of the Second USENIX Workshop on Electronic Commerce，1996年）。

② Fortezza，https://en.wikipedia.org/wiki/Fortezza（维基百科，检索于2014年6月30日）。

用decryption_failed警报。
- 这个版本引用包含了TLS扩展（RFC 3546）。

2.14.4　TLS 1.2

TLS 1.2于2008年8月发布。与TLS 1.1相比，它包含以下主要改进。

- 添加已验证加密支持。
- 添加对HMAC-SHA256密码套件的支持。
- 删除IDEA和DES密码套件。
- 虽然大部分扩展的实际文档还是在其他地方，但TLS将扩展和协议的主规格说明书进行了集成。
- 客户端可以使用一种新的扩展（signature_algorithms）来通报它愿意接受的散列和签名算法。
- 当使用TLS 1.2套件或者以协商协议是TLS 1.2为条件使用之前的套件时，在PRF中使用SHA256代替MD5/SHA1组合。
- 允许密码套件定义其自身的PRF。
- 使用单一散列代替用于数字签名的MD5/SHA1组合。默认使用SHA256，并且密码套件可以指定其自身使用的散列。签名散列算法以往是由协议强制指定，现在是散列函数式签名结构中的一部分，而且在实施启用中可以选择最佳算法。
- 密码套件可以显式指定Finished消息中的verify_data成员的长度。

第3章 公钥基础设施

有了公开密钥算法之后，我们就可以通过他人的公开密钥与其安全通信，但是还有一些悬而未决的问题：如何与那些从未谋面的人进行沟通？如何存储和吊销密钥？最重要的是，如何让现实世界中数以百万计的服务器、几十亿人和设备之间安全通信？这个问题非常的复杂，而公钥基础设施（public key infrastructure，PKI）就是为解决此问题而建立的。

3.1 互联网公钥基础设施

对大多数人来说，PKI就是互联网公钥基础设施。实际上PKI的含义更宽泛，因为其原本是为了别的用途而开发的。因此更准确的说法是，由PKIX工作组为适应PKI在互联网上的使用而提出的互联网公钥基础设施（Internet PKI）；另外一个最近常被用到的词是Web PKI，主要关注在浏览器上如何使用和验证证书。在本书中，除非某些特定场合确实有必要区分这两个概念以外，我说的PKI都表示互联网公钥基础设施。

PKI的目标就是实现不同成员在不见面的情况下进行安全通信，我们当前采用的模型是基于可信的第三方机构，也就是证书颁发机构（certification authority或certificate authority，CA）签发的证书。互联网PKI证书生命周期如图3-1所示。

- 订阅人

  订阅人（或者说最终实体）是指那些需要证书来提供安全服务的团体。

- 登记机构

  登记机构（registration authority，RA）主要是完成一些证书签发的相关管理工作。例如，RA会首先对用户进行必要的身份验证，然后才会去找CA签发证书。在某些情况下，当CA希望在用户附近建立一个分支机构时（例如在不同的国家建立当地登记中心），我们也称RA为本地登记机构（local registration authority，LRA）。实际上，很多CA也执行RA的职责。

- 证书颁发机构

  证书颁发机构（certification authority，CA）是指我们都信任的证书颁发机构，它会在确认申请用户的身份之后签发证书。同时CA会在线提供其所签发证书的最新吊销信息，这样信赖方就可以验证证书是否仍然有效。

- 信赖方

  信赖方（relying party）是指那些证书使用者。技术上来说，一般是指那些执行证书验证的网页浏览器、其他程序以及操作系统。他们是通过维护根可信证书库来执行验证的，这些证书库包含某些CA的最终可信证书（信任密钥，trust anchor）。更广泛地说，信赖方是指那些需要通过证书在互联网上进行安全通信的最终用户。

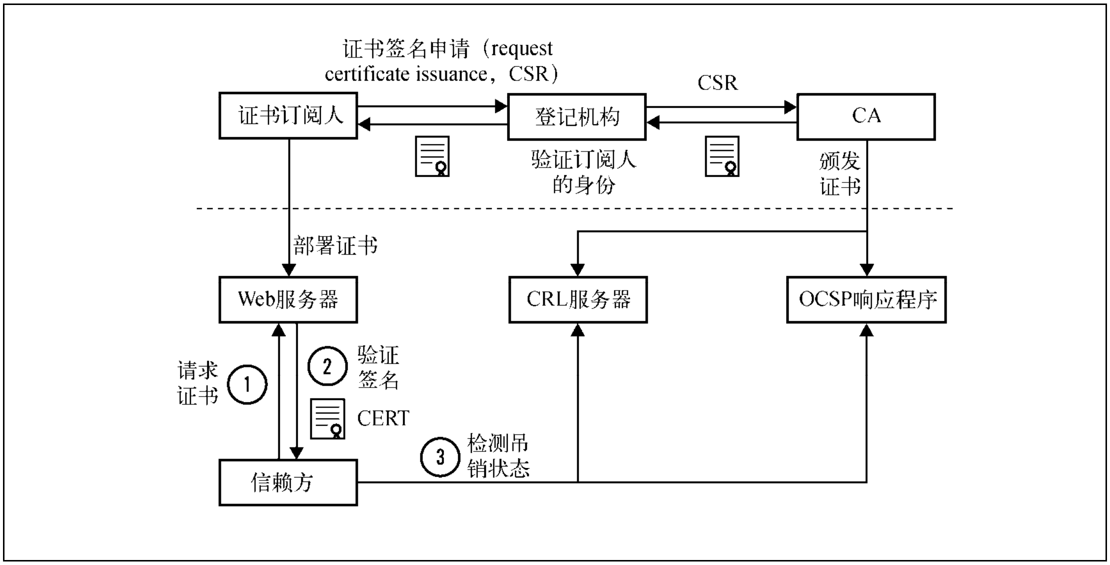


图3-1　互联网PKI证书生命周期

什么是信任？

讨论PKI的时候经常会用到**身份**（identity）、**权威**（authority）和**信任**（trust）等词汇，在这里它们的含义与平时的不一样，很容易让人将它们与现实生活中的含义混淆在一起。

对大多数证书来说，我们在与服务器通信的时候，对于服务器身份正确与否，证书其实只提供了有限的保证。只有EV证书会和线下身份进行绑定，但这里面依旧有很多其他因素，所以EV证书也并不意味着更加安全。

在PKI里面，**信任**只是技术层面上的概念，它仅仅意味着证书可以通过处于可信证书库中的某个CA的验证，但这并不意味着我们可以信任证书订阅人的一切。考虑一下这种情形：亚马逊网站虽然没有使用加密，但每天依旧有百万人访问并在上面进行购买。为什么？归根结底还是因为这些网站赢得了我们的信任。

3.2　标准

互联网公钥基础设施可以追溯到X.509，它是一种公钥基础设施的国际标准，最初是为了支持X.500而设计的。X.500是电子目录服务的标准，但是从来没有被广泛使用过；X.509经PKIX工作组的改造，适合在互联网上使用。[①]

下文节选自PKI宪章：

> PKIX工作组成立于1995年秋天，目标是建立支持基于X.509公钥基础设施的互联网标准。最开始PKIX主要是分析CCITT（之后是ITU-T）的X.509标准，不久之后，PKIX抛弃了ITU-T的工作，重新开始建立满足互联网需求的基于X.509的公钥基础设施。随着时间的流逝，这项工作成为了PKIX的工作重点，举个例子，大多数PKIX提交的RFC不再是ITU-T X.509文档的描述了。

PKIX工作组产出的最重要的文档是RFC 5280，它描述了证书格式、可信证书链的建立，以及证书吊销列表（CRL）的格式。[②]PKIX工作组在2013年10月结束了自己的使命。

> **注意**
> 互联网上最普遍的一点就是现实和标准从来都是大相径庭的，PKIX就如此。一部分是因为标准经常含糊不清而且满足不了现实的需求，另外想要预测未来技术的演变几乎不可能，所以每个人都有自己的实现方式。另外，主流的产品和代码库中经常会存在一些bug，极大地限制了很多技术在实际中的应用。本书中你会发现很多这样的例子。

CA/Browser论坛（CAB论坛）是由证书颁发机构、浏览器厂商以及其他有相关权益的团体自发形成的组织，目标是建立和推行证书颁发和处理的标准。[③]一开始，CAB论坛是为了确定增强型证书（EV证书）的颁发标准而创建的，在2007年EV证书诞生了。[④]CAB论坛最开始只是一些松散的组织，但是随后他们改变了关注的焦点并且在2012年改组。[⑤]同年，CAB论坛发布了*Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates*（《公共可信证书的颁发和管理基本要求》），简称为*Baseline Requirements*。[⑥]

CAB论坛虽然只列了大约40个证书颁发机构，但是*Baseline Requirements*适用于所有的证书

---

① PKIX工作组，http://datatracker.ietf.org/wg/pkix/charter/（IETF，检索于2014年7月16日）。

② RFC 5280: Internet X.509 Public Key Infrastructure Certificate and CRL Profile，http://tools.ietf.org/html/rfc5280（Cooper等，2008年5月）。

③ CAB论坛，https://cabforum.org（检索于2014年7月16日）。

④ EV SSL Certificete Cuidelines（CAB论坛，检索于2014年7月16日）。

⑤ 之所以会转变注意力是因为他们认识到还有很多严重的安全问题还没有被处理。在2011年，大大小小的CA出了好几个问题，给人的总体感觉就是PKI生态体系的安全性糟糕透了。甚至有人质疑这个生态还能存在多久。有了*Baseline Requirements*之后，CAB论坛解决了很多这类问题。

⑥ *Baseline Requirements*，https://cabforum.org/baseline-requirements/（CAB论坛，2014年7月13日检索）。

颁发机构；这份文件被纳入到针对证书颁发机构的WebTrust审计程序中[①]，而有些根证书库运营商（比如Mozilla）就明确要求CA必须符合这份标准。

另外一个相关组织是2012年9月份成立的IETF's Web PKI工作组，目的是描述PKI在Web上如何真正工作起来。[②]这个组织的目的是为Web PKI信任模型、吊销实践、证书中各字段和扩展的用法、证书吊销列表和OCSP响应制定参考文献。

3.3 证书

证书是一个包含公钥、订阅人相关信息以及证书颁发者数字签名的数字文件，也就是一个让我们可以交换、存储和使用公钥的壳。因此，证书成为了整个PKI体系的基础组成元素。

ASN.1、BER、DER和PEM

抽象语法表示法一（abstract syntax notation one，ASN.1）是支持复杂数据结构和对象的定义、传输、交换的一系列规则。ASN.1是为了支持不同平台的网络通信而设计，与机器架构以及语言实现无关。ASN.1于1988年最先在X.208中定义，最近的更新是在2008年推出的X.680系列文档中。

ASN.1以一种抽象的方式定义数据，如何编码数据则在另外一份标准里面。**基本编码规则**（basic encoding rules，BER）是第一个数据编码标准。X.509依赖于的**唯一编码规则**（distinguished encoding rules，DER）是BER的子集，只允许一种方式编码ASN.1的值，这种唯一性对密码学尤其是数字签名的使用非常关键。PEM（privacy-enhanced mail的简写，在此上下文中无含义）是DER使用Base64编码后的ASCII编码格式。ASN.1比较复杂，除非你在做密码相关的开发工作，一般情况下不需要与它打交道。

大多数的证书都是PEM格式（因为这种格式更容易发送、复制和粘贴），当然有时候你也许会遇到DER格式。后面的章节我们会提到如何使用OpenSSL x509命令进行不同格式之间的转换。

如果你想看看ASN.1的样子，可以随意下载一张证书，然后使用在线ASN.1解码器来查看ASN.1的结构[③]。

邮电

3.3.1 证书字段

证书由一些字段组成，在版本3里还包括一些扩展。从表面上来看，证书的结构是扁平而线性的，但是一些字段还包括了别的结构。

① WebTrust Program for Certification Authorities，http://www.webtrust.org/homepage-documents/item27839.aspx（Webtrust，检索于2014年5月25日）。

② Web PKI OPS，http://datatracker.ietf.org/wg/wpkops/charter/（IETF，检索于2014年5月25日）。

③ ASN.1 JavaScript decoder，http://lapo.it/asn1js/（Lapo Luchini，检索于2014年5月24日）。

- 版本

  证书一共有3个版本号，分别用0、1、2编码表示版本1、版本2和版本3。版本1只支持简单的字段，版本2增加了两个标识符（新增的字段），而版本3则增加了扩展功能。现在大部分的证书都采用版本3的格式。

- 序列号

  在一开始，序列号只要是正整数即可，是每个CA用来唯一标识其所签发的证书。但是在出现了针对证书签名的预选前缀攻击之后，序列号增加了更多的要求来防止此类攻击（在4.5节中有更详细的介绍）；现在序列号需要是无序的（无法被预测）而且至少包括20位的熵。

- 签名算法

  这个字段指明证书签名所用的算法，需要放到证书里面，这样才能被证书签名保护。

- 颁发者

  颁发者（issuer）字段包括了证书颁发者的可分辨名称（distinguished name，DN），这个字段比较复杂，根据不同的实体会包含许多部分。举例来说，Verisign根证书的可分辨名称是`/C=US/O=VeriSign, Inc./OU=Class 3 Public Primary Certification Authority`；它包括了国家、组织和组织单位三个部分。

- 有效期

  证书的有效期包括开始日期和结束日期，在这段时间内证书是有效的。

- 使用者

  使用者是实体的可分辨名称，和公钥一起用于证书的签发。在自签名证书里，使用者（subject）和颁发者（issuer）字段的可分辨名称是一样的。在最开始，可分辨名称里面的公用名（common name，CN）主要用于服务器主机名（例如`/CN=www.example.com`用于www.example.com域名的证书），但是如何为一个证书匹配多个主机名就变得比较麻烦了。如今，使用者字段已经废弃，转而使用使用者可选名称扩展。

- 公钥

  这个字段包含了公钥，以使用者公钥信息（subject public-key info）结构呈现（主要是算法ID，可选参数以及公钥本身）。公钥算法在RFC 3279里面有具体说明。[1]

> **注意**
>
> 在版本2里面新增了两个字段，分别是**颁发者唯一 ID**（issuer unique ID）和**使用者唯一ID**（subject unique ID）。在版本3里面使用**授权密钥标识符**（authority key identifier）和**使用者密钥标识符**（subject key identifier）扩展代替了版本2的颁发者唯一ID和使用者唯一ID。

① RFC 3279: Algorithms and Identifiers for the Internet X.509 PKI and CRL Profile，http://tools.ietf.org/html/rfc3279（Polk等，2002年4月）。

3.3.2 证书扩展

为了让原本死板的证书格式变得更加灵活，版本3引入了证书扩展。每一个扩展都包括唯一的对象标识符（object identifier，OID）、关键扩展标识器以及ASN.1格式的值。如果将某个扩展设置为关键扩展，那么客户端必须能够解析和处理这个扩展，否则就应该拒绝整张证书。

- 使用者可选名称

  原本使用者证书字段（更准确地说是其中的通用名部分）是用来将身份信息和公钥绑定在一起的。而在实际使用的时候发现使用者字段不够灵活，只能支持与一个主机名进行绑定，无法同时处理多个身份信息。使用者可选名称扩展就是为了替换使用者字段，它支持通过DNS名称、IP地址和URI来将多个身份绑定在一起。

- 名称约束

  名称约束扩展可以限制CA签发证书的对象，这样命名空间就在可控范围内。这个功能非常有用，例如，它允许一个组织可以拥有一个二级CA，而这个CA只能签发这个公司所拥有的那些域名下的证书。有了这个限制，这类CA就不会影响整个生态系统了（例如，CA不能签发任意网站的证书）。RFC 5280要求将这个扩展设置为关键扩展，但是实际情况是，大部分CA都将其设置为非关键扩展，而*Baseline Requirements*也明确表示允许这么处理。主要原因是有一些产品无法解析名称限制这个扩展，如果标记为关键扩展，就会导致这些产品拒绝此类证书。

- 基础约束

  基础约束扩展用来表明证书是否为CA证书，同时通过路径长度（path length）约束字段，来限制二级CA证书路径的深度（例如限制CA证书是否可以签发更深一层的CA证书以及能签发多少层）。理论上，所有的CA证书都必须包含这个扩展；而实际情况是，有一些使用版本1协议的根证书还在使用中，它们是没有任何扩展功能的。

- 密钥用法

  该扩展定义了证书中密钥可以使用的场景，这些场景已经定义好了，可以通过设置来让证书支持某个场景。例如CA证书一般都设置了证书签名者（certificate signer）和CRL签名者（CRL signer）。

- 扩展密钥用法

  为了更加灵活地支持和限制公钥的使用场景，该扩展可以通过OID支持更多的场景。例如最终实体证书一般都拥有`id-kp-serverAuth`和`id-kp-clientAuth`两个OID，代码签名证书使用`id-kp-codeSigning` OID等。

  虽然RFC 5280表明扩展密钥用法（extended key usage，EKU）只能用在最终实体证书上，但在实际中我们会看到中间CA也会带上这个扩展，从而让它签发出来的证书也带上这个限制。[①]*Baseline Requirements*还特别要求在使用EKU限制中间CA的同时还应当考虑使用

① Bug 725451: Support enforcing nested EKU constraints, do so by default，https://bugzilla.mozilla.org/show_bug.cgi?id=725351（Bugzilla@Mozilla，2014年2月8日报道）。

名称限制。

- 证书策略

  该扩展包含了一个或多个策略，每个策略都包括一个OID和可选限定符（qualifier）。限定符一般包括一个URI，从这个URI可以获得完整的策略说明。*Baseline Requirements*要求每一张最终实体证书需要包括至少一条策略信息，来表明该证书是在何种条款下签发的。另外这个扩展还能表明证书的验证类型。

- CRL分发点

  该扩展用来确定证书吊销列表（certificate revocation list，CRL）的LDAP或者HTTP URI地址。按照*Baseline Requirements*，每一张证书都至少需要包括CRL或者OCSP吊销信息。

- 颁发机构信息访问

  颁发机构信息访问扩展表明如何访问签发CA提供的额外信息和服务，其中之一就是OCSP响应程序的HTTP URI地址。信赖方可以使用这个服务来实时检测证书的吊销信息。另外还有一些证书包含了签发CA的URI地址，有了这个地址，即便服务器返回的证书链中缺少了签发CA的证书，客户端也可以通过下载签发CA重新构建证书链。

- 使用者密钥标识符

  该扩展包含了唯一的值，可以用来识别包含特别公钥的证书，一般建议使用公钥本身来建立这个标识符（例如通过散列）。所有的CA证书都必须包含这个扩展，并且它的值要与CA所签发出来的证书上的授权密钥标识符的值一样。

- 授权密钥标识符

  这个扩展的内容是签发此证书的CA的唯一标识符，通常用于在构建证书链时找到颁发者的证书。

RFC 5280还定义了几个很少使用到的扩展，它们分别是增量CRL分发点、禁止任意策略、颁发者可选名称、策略限制、策略映射、使用者目录属性、使用者信息访问。

3.4　证书链

在大多数情况下，仅仅有最终实体证书是无法进行有效性验证的，所以在实践中，服务器需要提供证书链才能一步步地最终验证到可信根证书，如图3-2所示。证书链的使用可能出于安全、技术和管理等方面的原因。

- 保证根证书安全

  根CA不仅对拥有它的组织很重要，对整个生态来说同样至关重要。首先是它有巨大的经济价值，因为很多根证书库已经不再更新了，所以以前那些已经被广泛使用的根CA是不可替代的。再者，如果根CA的私钥被泄露，那么就可以签发任意域名的虚假证书。另外如果根CA会被吊销掉，所有使用这个CA签发出来的证书的网站都会无法访问。

  现在仍然还有很多CA直接使用他们的根证书直接签发最终实体证书，这其实是非常危险的。*Baseline Requirements*限制所有的根证书密钥只能由人手动执行命令（自动化是不允

许的），也就是说根证书密钥必须离线保存。虽然还有很多依旧在使用的旧系统有这个漏洞，但是直接由根证书签发最终实体证书是不允许的。

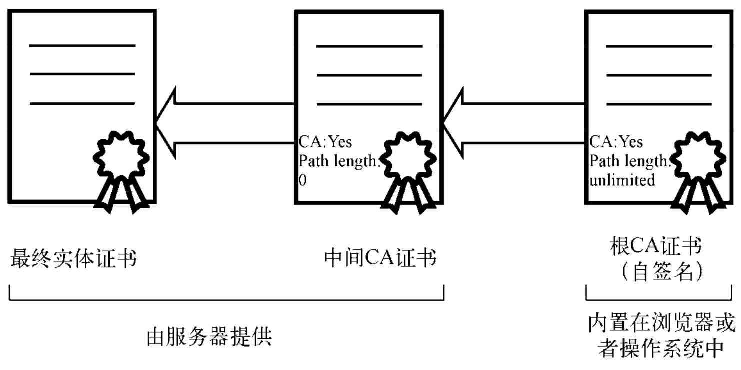


图3-2　证书链

- 交叉证书

  交叉证书是可以让新的CA立即投入运营的唯一方式。因为想要在短期内让新的根证书部署得足够广泛是不可能的，所以新的CA都会找已经进行广泛内置的CA对他们的根密钥进行签名。随着时间的流逝，那些老的设备会逐渐淘汰掉，新的CA才能最终独立使用。

- 划分

  CA在将它的操作分散给很多二级CA的同时有可能会带来更多的风险。例如不同的二级CA用于签发不同的证书类型，或者由不同的业务部门使用。与根证书不同的是，二级CA一般都是在线的，而且使用自动化系统签发证书。

- 委派

  还有一些情况是CA希望给外部其他组织签发一个二级CA。例如一家大的公司可能希望可以自己签发它所拥有的那些域名的证书（这种方式通常比去维护私有CA，并确保所有设备都内置这个CA来说成本更低）。有时候一些组织希望能够完全保管这个二级CA，这时候CA就会从技术上限制这个二级CA只能签发某些域名；其他情况下CA依旧可以控制二级CA签发出来的证书。

服务器一次只能提供一条证书链，而实际上可能存在多条可信路径。以交叉证书为例，一条可信路径可以一直到CA的主要根证书，另外一条则是到可选根证书上。CA有时候会为同样的密钥签发多张证书，例如现在最常使用的签名算法是SHA1，因为安全原因正在逐步迁移到SHA256，CA可以使用同样的密钥签发出不同签名的新证书。如果信赖方恰好有两张这样的证书，那么就可以构建出两条不同的可信路径。

路径的建立让整个事情变复杂了，而且导致了很多问题。服务器必须提供完整并且有效的证

书链，但是因为人员的疏忽或者各种各样的配置问题，导致证书链的配置总是有问题（例如需要在不同的地方配置服务器证书和证书链剩余的部分）。根据SSL Pulse的数据，大概有5.9%的服务器配置的证书链是不完整的。[①]

另外，因为标准的模糊、不完整以及相互矛盾，客户端在建立和验证路径上不可避免地出现了很多安全问题。历史上有很多验证库在验证签发CA属于哪个根CA这类简单问题上都出现过问题。如今最常使用的那些库并不是从一开始就安全的，也是经历过各种问题，打上了各种补丁后才慢慢经受住了实践的考验。更多的例子可以参考6.1节。

3.5　信赖方

信赖方为了能够验证证书，必须收集信任的所有根CA证书。大多数的操作系统都提供一个根证书库，从而在一开始启动的时候就能够建立信任。几乎所有的软件开发者都重用了底层操作系统提供的根证书库，唯一的例外是Mozilla，为了保证不同平台的兼容性，它维护了自己的根证书库。

- Apple

  Apple维护的根证书库主要是给iOS和OS X平台使用[②]，如果某个CA希望加入到Apple的根证书库里面的话，就需要通过权威机构审计并且对Apple的客户有商业价值。
- Chrome

  在Linux上，Chrome使用Mozilla的根证书库（通过NSS网络库），除此之外Chrome都是依赖操作系统提供的证书库。即便如此，Chrome在底层设施的基础上还额外增加了很多策略。[③]举例来说：(1) Chrome增加了根证书黑名单；(2) 增加了能够签发EV证书的CA列表；(3) 要求所有EV证书从2015年2月开始，必须支持证书透明度。
- Microsoft

  Microsoft维护的根证书库主要是给Windows桌面版、服务器版以及移动手机平台使用。[④]同样，如果要加入，需要至少一年的审计并且提供一份能够为Microsoft的用户群带来商业价值的说明。
- Mozilla

  Mozilla为自己的产品维护了一个公开透明的根证书库[⑤]并且大部分Linux版本都使用

① SSL Pulse，https://www.trustworthyinternet.org/ssl-pulse/（SSL Labs，检索于2014年7月）。

② Apple Root Certificate Program，https://www.apple.com/certificateauthority/ca_program.html（Apple，检索于2014年5月25日）。

③ Root Certificate Policy，http://www.chromium.org/Home/chromium-security/root-ca-policy（Chrome安全，2014年5月25日）。

④ Introduction to The Microsoft Root Certificate Program，http://social.technet.microsoft.com/wiki/contents/articles/3281.introduction-to-the-microsoft- root-certificate-program.aspx（Microsoft，检索于2014年5月25日）。

⑤ Mozilla CA Certificate Policy，https://www.mozilla.org/en-US/about/governance/policies/security-group/certs/policy/（Mozilla，检索于2014年5月25日）。

Mozilla的根证书库。在`mozilla.dev.security.policy`列表和Mozilla的bug跟踪系统上经常会有一些针对制定策略的热门讨论话题。

所有的根证书库都要求CA通过专门为证书颁发机构设计的独立审计。要签发DV和OV证书，必须通过下面至少一项审计。

- 针对证书颁发机构的WebTrust审计①
- ETSI TS 101 456
- ETSI TS 102 042
- ISO 21188:2006

要想签发EV证书，还必须通过由WebTrust运营的EV证书审计程序。

3.6 证书颁发机构

证书颁发机构（certification authority，CA）是当前互联网信任模型最重要的部分，他们可以签发任何域名的证书，所以是非常权威的。表面上看来似乎是一门稳赚的生意，前提是你的根证书要内置到尽可能多的设备中。那么具体需要做什么才能成为一个公开的CA？

(1) 建立CA组织。
  a. 在PKI和CA的运营上非常专业。
  b. 需要设计一个健壮、安全、隔离的网络，以便在支持商业运作的同时，能够保护根证书以及二级证书密钥的安全。
  c. 支持证书生命周期管理流程。
  d. 符合*Baseline Requirements*的规定。
  e. 符合EV SSL证书指导规范。
  f. 提供全球化的CRL和OCSP基础服务。

(2) 符合当地法律，这意味着可能需要按照当地的法规要求获取相应许可证。

(3) 通过根证书库认可的那些审计。

(4) 将你的根证书内置到尽可能多的设备、软件中。

(5) 找个已经内置的CA完成交叉证书，之后就可以开始运作了。

在很长一段时间里，那些很早就进入这个市场的证书机构可以非常容易地出售证书。但是因为竞争越来越激烈，DV证书的价格下降得非常多，所以仅靠卖DV证书已经越来越难挣钱了。此外，如果DNSSEC和DANE得到广泛使用，DV证书的日子也就到头了。因此现在CA正在转向更小众、但可能更有赚头的EV证书市场和相关服务。

① 证书颁发机构原则和标准，http://blog.spiderlabs.com/2012/02/clarifying-the-trustwave-ca-policy-update.html（WebTrust，检索于2014年5月25日）。

3.7　证书生命周期

证书的生命周期在订阅人准备证书签名申请（certificate signing request，CSR）文件，并将它提交给所选CA的时候就开始了。CSR文件的主要目的是携带公钥信息，并且证明订阅人拥有对应的私钥（通过签名来证明）。CSR还设计携带额外的元数据，但实际中并非所有的都用到了。CA一般都会覆盖CSR文件的一些内容并且将其他信息内置到证书里面。

CA会根据不同类型的证书申请，执行不同的验证流程。

- 域名验证

  域名验证（domain validated，DV）证书需要CA验证订阅人对域名的所有权之后才能进行签发。大多数情况下CA会发送一封确认邮件给域名的管理邮箱，管理员通过之后（按照邮件里面的步骤和链接）CA就会签发证书。如果无法通过邮件确认，那么CA通过别的通信手段（例如电话或者邮寄信件）或者合理的方式证明订阅人对域名的所有权之后就可以签发证书。签发IP地址证书的步骤也是类似的。

- 组织验证

  组织验证（organization validated，OV）证书会对身份和真实性进行验证。直到采用了*Baseline Requirements*之后，OV证书的验证流程才标准化起来，但是在如何签发OV证书以及如何将这些信息编码到证书中等方面，依旧存在很多前后不一致的情况。

- 扩展验证

  扩展验证（extended validation，EV）证书以更加严格的要求验证身份和真实性。它是为了解决OV证书缺乏的前后一致性而引入的，所以EV证书的验证流程非常详细，几乎不会出现前后不一致的情况。

DV证书的签发是全自动的，所以非常快，它的签发时间主要取决于DNS管理员确认邮件所需的时间；而EV证书则相反，可能需要几天甚至几周才能拿到。

> **注意**
> 攻击者有可能会向CA提交那些知名度高的域名的证书签名申请文件。因此CA需要具备一些高风险的域名列表，对于这类域名的证书申请采用人工方式进行验证，否则就拒绝签发证书。这也是*Baseline Requirements*里的规定。

CA在验证成功之后就会签发证书。除了证书本身，CA还会提供所有的中间证书，从而构建证书链到对应的根证书上，当然对一些主流的平台也会介绍如何进行配置。

在证书有效期范围内，申请者可以在他们的生产环境中使用该证书。如果证书对应的私钥泄露了，那么就需要吊销证书。这个过程和证书的签发有点类似。有一种说法叫作补签证书，不过从技术角度看，不存在补签证书一说：如果一张证书被吊销了，那么就需要按照证书申请签发流程换一张新证书。

3.8 吊销

当出现私钥泄露或者不再需要使用的时候，我们就需要吊销证书。但是这里存在误用的风险。吊销协议和流程的设计是为了确保证书是有效的，否则就需要将吊销情况通知信赖方。现在有下面两种证书吊销标准。

- ❑ 证书吊销列表

  证书吊销列表（certificate revocation list，CRL）是一组未过期、但是却已经被吊销的证书序列号列表，CA维护了一个或多个这样的列表。每一张证书都需要在CRL分发点（CRL distribution point）扩展中包含对应的CRL地址。CRL最大的问题在于它越来越大，实时查询起来会非常慢。
- ❑ 在线证书状态协议

  在线证书状态协议（online certificate status protocol，OCSP）允许信赖方获得一张证书的吊销信息。OCSP服务器通常称为OCSP响应程序，OCSP响应程序的地址编码在颁发机构信息访问（authority information access，AIA）证书扩展中。OCSP支持实时查询并且解决了CRL最大的缺点，但是并没有解决所有的吊销问题：因为OCSP的使用带来了性能、隐私方面的问题和新的漏洞。其中一部分问题可以通过OCSP stapling技术来解决，它允许服务器在TLS握手的过程中直接嵌入OCSP响应。

3.9 弱点

从严格的安全视角来看，互联网PKI存在许多大大小小的弱点，在本节中我会将它们列出来。当然，在讨论这些弱点之前，我们需要先看看它的历史。在1995年，也就是安全Web刚刚兴起的时候，互联网完全和现在不一样，而且也没有现在这么重要。在那个时候，我们需要加密协议才能开办电子商务，才能赚钱。如今我们已经有了非常不错的电子商务，但是我们想要的更多。对一些组织来说，加密真的是生死攸关。

但是我们如今拥有的体系还是之前设计的，为的只是保证电子商务操作足够安全。从更宽泛的概念上来讲，这个体系提供了我称为商业安全（commercial security）的保障。用相对很少的钱就能达到这样的安全，它可以让网站跑起来更快，可以允许一些不安全的实践，而且几乎不影响用户。它由那些追逐利润的CA厂商，以及只注重扩大市场份额的浏览器供应商所控制，而他们从未将安全列入最高优先级。当然也不能完全归咎于他们，只有我们这些最终用户行动起来，提出更多的安全需求，他们才会作出改变。

仅有CA是无法成功的。每年，数百个CA会签发出几百万张证书来让整个世界正常运转，错误率非常低。当然在安全上没有想象得那么美好，但是整个体系运转得还挺好。尽管如此，还是有很多的订阅人因为要付费购买证书而对CA产生不满。他们不想掏钱，同时却又希望拥有绝对的安全。

事实上，人们是无法从当前这个不愿意破旧立新的生态体系中获得绝对安全的，无论这是好

还是坏。尽管如此，问题正在得到修复，后面会进行介绍。好了，现在我们看看这些缺陷。

- 未经域名所有者授权就可以签发证书

  从原理上看，我们现在最大的问题是：任何CA都可以在未经域名所有者授权的情况下签发该域名的证书。这里关键的问题就是当前没有任何技术手段可以确保CA不会产生疏漏和安全过错。在只有几个CA的那个年代，这可能不是个问题，但是现在已经有了几百个CA，这就成了一个大问题。如今整个PKI体系的安全级别等同于其中最薄弱的环节，而现在我们已经有了很多潜在的薄弱环节。所有的CA都需要经过审计，但是审计的质量同样存疑。例如一家荷兰的CA公司DigiNotar，虽然经过审计，但是在2011年，其整个安全体系依旧被完全入侵。

  那么接下来的问题就是，我们能否信任CA可以做好他们自己的工作并且能够考虑公众利益？我们应该允许哪些CA有权利在缺少监督的情况下签发证书？有时候我们有正当的理由去担心这些CA会将他们的商业利益置于我们的安全需求之上。例如，TrustWave在2012年承认签发了一个二级CA用于流量审查，它会给任意网站签发伪造的证书。[①]虽然TrustWave是唯一公开承认签发这类证书的CA，但还有很多传言说这种情况并不少见。

  很多人担心政府部门会滥用这个体系，伪造签发任意域名的证书。我们真的可以确定一些CA后面没有政府的背景吗？即便真的没有，我们又如何确保他们不会被迫做政府要求的事情？唯一无法确认的是，不知道政府在商业CA中渗透得有多深。

- 缺少信任灵活度

  还有一个理论上的问题是缺少信任灵活度。信赖方维护了包含一定数量CA证书的根证书库，这样CA要么是完全可信，要么完全不可信，没有中间情况。理论上，信赖方是可以从根证书库里移除CA的，但实际上，只有出现严重不称职或者安全泄露问题，或者CA体量很小时，才会出现上面这种情况。一旦CA签发了大量的证书，就会出现大而不倒的情况。

  某些象征性的惩罚还是有可能的。例如，过去有信赖方吊销过某些不称职CA的EV证书权限。另外一个想法（从未尝试过）就是，对于有过恶劣行为的CA，不再信任他们未来签发的证书，只信任他们已经签发的证书。

- 弱域名验证

  DV证书是通过不安全的WHOIS协议获取域名信息，然后基于域名所有者信息来签发证书的。此外，验证过程经常是用邮件沟通，本身也是不安全的。如果攻击者劫持了域名或者获得了关键邮箱的访问权限，那么要获得一张伪造的DV证书会非常容易。另外还可以通过窃听CA端的网络来攻击CA的验证过程。

- 吊销不生效

  吊销不生效的情况比较普遍。我们发现在2011年有几个CA的吊销失效了。在每个案例中，

---

① Clarifying The Trustwave CA Policy Update，http://blog.spiderlabs.com/2012/02/clarifying-the-trustwave-ca-policy-update.html（Trustwave，2012年2月4日）。

信赖方不得不打补丁或者使用他们自有的黑名单通道来吊销错误签发的证书。

有两个原因说明这是有必要的。首先在每个系统之间传播吊销信息会有延迟。*Baseline Requirements*允许CRL和OCSP的信息有效期最多可以是10天（中间证书则是12个月）。也就是说吊销信息的传播需要至少10天的时间。第二个问题是当前所有浏览器都已实现的软失败（soft-fail）策略；它们会尝试获取吊销信息但会忽略所有的失败。主动网络攻击者可以很容易地阻断OCSP请求，然后让其无限期地使用本已无效的证书。

浏览器厂商因此决定停止吊销检测。CRL首当其冲，紧接着是OCSP。目前的趋势是利用专有的机制，先查询少数中间证书的可靠吊销信息，再查询其他的吊销信息。未来的解决方法可能是提供一种新的吊销信息查询机制，但是该方法现在还毫无进展。

你可以在5.10节中了解到有关本主题的完整介绍。

- **证书警告让加密意图完全失效**

  互联网PKI（准确说是Web PKI）最大的失败可能是证书验证的松散实现。很多库和应用完全绕过了验证过程。浏览器虽然会检测证书，但是当遇到无效证书时，它们却又允许用户忽略呈现给它们的警告信息。根据一些统计估计，大概有30%~70%的用户会点击跳过这些警告，这让我们的加密意图完全失效。最近，一种称为HTTP严格传输安全（HTTP strict transport security，HSTS）的新标准出现了，它要求所有实现它的浏览器用错误替换警告，并且不允许忽略错误。

3.10 根密钥泄露

攻击PKI最好的方法之一是直接对根证书下手。如果是政府部门，可以直接要求该国的CA交出他们的私钥。如果觉得这种行为存在争议或者非常危险，那么只要有一些预算（比如一百万美元左右）就可以自己创建一个新的CA品牌，并且将其根证书内置到所有的证书库里面。他们甚至可能觉得没有必要去好好运作一个CA以便进行掩饰，因为有很多根证书从来没有签发过最终实体证书。

这种攻击互联网PKI的手段在之前的很多年都是行之有效的，但是从近两年开始，人们对整个生态系统发生的事情越来越关注。浏览器建立了证书跟踪的插件，它们会在新证书出问题的时候警告用户。Google在非常流行的Chrome里面实现了公钥钉扎（public key pinning）。电子前线基金会对他们的HTTPS Everywhere插件进行了扩展，增加了对根证书使用的监控。[①]

另外一种（在过去和现在都）不这么复杂的方式是打破现存的根证书和中间证书。如果你有权限访问中间证书对应的私钥，那么就可以签发任意的证书。为了最好的效果（尽可能不被发现），签发伪造证书的CA最好与真正的CA一样。很多站点，特别是那些大型站点，同一时间可能会运营着多张证书。如果签发CA是一样的，那么要如何区分伪造证书和真实证书呢？

2003年（已经是10多年前的事了），Shamir和Tromer估计花费1000万美元特别建造的机器可

① HTTPS Everywhere，https://www.eff.org/https-everywhere（电子前线基金会，检索于2014年7月3日）。

以在1年内破解1024位的密钥（还需要额外的2000万美元的设计和开发费用）。[1]按照那些可能已经公开的伪造证书来计算，这对国家机构来说其实是非常廉价的。这些机构通常会花费几十亿美元在有兴趣的项目上。2013年，Tromer估计成本下降到只要100万美元。[2]

从Tromer的估计来看，有理由相信所有1024位长度的密钥已经被不同国家的多个政府机构破解了。

> **注意**
> 对中间证书来说，另外一个攻击坐标是弱SHA1签名。在遇到碰撞攻击和前映像攻击的时候，SHA1最多只能提供80位和160位的安全。中间证书更容易成为攻击的目标，因为它们不像根证书那样引人注意。

在某些情况下，有理由相信最终实体证书也被盯上了。例如Google在2013年把所有的1024位证书都替换掉了。[3]

奇怪的是，既然知道了攻击弱证书的成本很低，但我们还在使用弱根证书。Mozilla计划在2013年底移除这类根证书，[4]但是因为可能会造成很大的影响，所以他们延期了。如果要跟踪他们的进展，可以查看bug #881553。[5]在写这本书的时候，也就是2015年7月份，Mozilla计划在2015年末移除剩余的弱根证书。[6]

3.11　生态系统评估

在2010年之前，很少能在公开场合听到有关PKI生态体系的信息。2010年起，对PKI生态系统的主动扫描和监控开始了。在2010年7月份的Black Hat USA上，我发表了对1.2亿域名的调查结果，分析了证书以及TLS服务器的安全。[7]几天之后，电子前线基金会（Electronic Frontier Foundation，EFF）在DEFCON上宣布了他们对整个IPv4的地址段进行调查的项目SSL Observatory。[8]他们的关

① On the Cost of Factoring RSA-1024，http://tau.ac.il/%7Etromer/papers/cbtwirl.pdf（Shamir和Tromer，2013年）。

② Facebook’s outmoded Web crypto opens door to NSA spying，http://www.cnet.com/uk/news/facebooks-outmoded-web-crypto-opens-door-to-nsa-spying/（CNET，2013年6月28日）。

③ Google certificates upgrade in progress，http://googledevelopers.blogspot.co.uk/2013/07/google-certificates-upgrade-in-progress.html（Google Developers Blog，2013年7月30日）。

④ Dates for phasing out MD5-based signatures and 1024-bit moduli，https://wiki.mozilla.org/CA:MD5and1024（MozillaWiki，检索于2014年7月3日）。

⑤ Bug #881553：Remove or turn off trust bits for 1024-bit root certs after December 31, 2013，https://bugzilla.mozilla.org/show_bug.cgi?id= 881553（Bugzilla@Mozilla，2013年6月10日报道）。

⑥ Bug #1156844 - Turn off trust bits for Equifax Secure Certificate Authority 1024-bit root certificate，https://bugzilla.mozilla.org/show_bug.cgi?id=1156844（Bugzilla@Mozilla，2015年4月21日报道）。

⑦ Internet SSL Survey 2010 is here!，http://blog.ivanristic.com/2010/07/internet-ssl-survey-2010-is-here.html（Ivan Ristić，2010年7月29日）。

⑧ The EFF SSL Observatory，https://www.eff.org/observatory（电子前线基金会，检索于2014年5月26日）。

注点主要在证书上，但是最大的贡献是公开了所有的数据，点燃了很多人的想象力，也引领更多人进行了类似的扫描。之后EFF还发布了“分布式SSL观察报告”，[①]通过他们的HTTPS Everywhere浏览器插件收集了证书链的信息，但是到现在他们还没有发表任何报告。

2011年，Holz等发布了他们的数据集，数据来源于第三方对整个IPv4段的扫描，他们自己对Alexa排名前100万的域名的安全服务器的扫描，以及对他们的研究网络进行被动流量监控。[②]他们也公开发表了数据。

2012年4月，SSL Labs启动了SSL Pulse项目，该项目从Alexa排名前100万的域名中抓取出150 000个最流行的安全网站，然后每月定期进行扫描。[③]

同年，国际计算机研究中心（International Computer Science Institute，ICSI）宣布他们的ICSI Certificate Notary项目，该项目会监控10个合作伙伴的实时网络流量。[④]他们的报告非常有趣，展现了现实生活中证书以及加密参数。他们还将整个PKI生态系统和各CA之间的可视化关系记录保留在他们的信任树（Tree of Trust）中。[⑤]

到2013年为止，最全面的研究由Durumetric等发表，他们对整个互联网进行了长达14个月的110次扫描。[⑥]为实现该项目，他们开发了一个专门的工具ZMap，现在已经开源了。他们所有的数据都已经在网上公开。[⑦]如果你想获取原始数据，在同一个网站上还可以找到Rapid7发布的月度证书扫描数据。[⑧]

调查显示没有致命的问题，但是这些报告为了解PKI生态系统提供了很好的视角，同时突出了一些重要的问题。例如公众其实不知道CA会定期给私网IP地址（任何人可以在他们的内部网络使用）以及内部不完整的域名（例如localhost、mail、intranet等）颁发证书。几年之后，大规模的扫描越来越正常，还有一些成就，例如证书透明度（3.12节中将讨论），它依赖于所有可用公开证书。2014年2月，Microsoft宣布他们在Internet Explorer 11使用的自动远程控制技术会开始收集证书数据。[⑨]他们希望可以用这些信息来快速检测到针对IE浏览器用户的攻击。

同样在2014年2月，Delignat-Lavaud等发布了证书供应商对CAB论坛指导方针的遵守情况的

① HTTPS Everywhere & the Decentralized SSL Observatory，https://www.eff.org/deeplinks/2012/02/https-everywhere-decentralized-ssl-observatory（Pter Eckersley, 2012年2月29日）。

② The SSL Landscape - A Thorough Analysis of the X.509 PKI Using Active and Passive Measurements，https://pki.net.in.tum.de/node/15（Holz等，互联网评估会议，2011年11月）。

③ SSL Pulse，https://www.trustworthyinternet.org/ssl-pulse/（SSL Labs，检索于2014年7月19日）。

④ ICSI Certificate Notary，http://notary.icsi.berkeley.edu（ICSI，检索于2014年7月19日）。

⑤ ICSI SSL Notary: CA Certificates，http://notary.icsi.berkeley.edu/trust-tree/（ICSI，检索于2014年5月26日）。

⑥ Analysis of the HTTPS Certificate Ecosystem，http://conferences.sigcomm.org/imc/2013/papers/imc257-durumericAemb.pdf（Durumetric等，互联网评估会议，2013年10月）。

⑦ University of Michigan · HTTPS Ecosystem Scans，https://scans.io/study/umich-https（互联网扫描数据存储库，检索于2014年5月26日）。

⑧ Rapid7 · SSL Certificates，https://scans.io/study/sonar.ssl（互联网扫描数据存储库，检索于2014年5月26日）。

⑨ A novel method in IE11 for dealing with fraudulent digital certificates，http://blogs.technet.com/b/pki/archive/2014/02/22/a-novel-method-in-ie11-for-dealing-with-fraudulent-digital-certificates.aspx（Windows PKI Blog，2014年2月21日）。

评估。[1]结果显示对EV证书的遵守情况最好，在引入*Baseline Requirements*之后，不但严格要求，还有所进步。

我们对互联网PKI有多了解？

证书供应商每年签发数百万张证书。真正的数字仍在持续增长，但我们估计大概有五百万张证书在用。除此之外还有非常多的内部和自签名证书，但是因为都是在内部网络使用，没有人可以评估出到底有多少。

还不清楚现在到底有多少个CA，从主要的根证书库估计，大概有超过100个共同的根，但是很多CA拥有不止一个根。还有超过一千个二级CA证书，但是它们主要用于管理；还不清楚到底有多少组织有能力直接签发证书，我们知道排名前10的根占据了90%的市场，这些大公司分别是Symantec、Godaddy、Comodo、GlobalSign、Digicert、StartCom和Entrust。

3.12　进步

多年以来，我们看到了很多改变PKI的提案，特别是在2011年。好几家CA在那一年受到入侵，让我们觉得整个互联网都要瓦解了。这里我会简单介绍一下这些提案，不过它们中的大部分还未最终确定。还有一些提案在宣布之后没有什么进展。唯一的例外是钉扎（pinning）和DANE，这些技术几乎可以在实际中使用，所以第10章会进行详细介绍。

- 透视

  透视[2]首先在TLS验证过程中引入独立公证人的概念。由客户端推荐可信的公证人，而不是直接单独对证书颁发机构资质的有效性作出决定。从不同的有利位置去访问同一个服务器可以击败客户端附近发生的攻击。公证人可以对服务器进行一段时间的跟踪来击败更高级别的攻击。透视在2008年启动，目前仍在运作。

- 收敛

  收敛[3]是透视的一个分支，并且在某些实现方面有所提高。为增加隐私，发给公证人的请求需要经过多个服务器，这样公证人只知道客户端的身份，但是不知道请求的内容。为提高性能，站点证书会被缓存一段时间。在2011年收敛刚被提出的时候势头很猛，但是从2013年开始就很少见有人提它了。最可能的原因是浏览器无法提供足够的API来支持想要做信任决策的插件。

- 公钥钉扎

① Web PKI: Closing the Gap between Guidelines and Practices，http://research.microsoft.com/pubs/206278/ndss.pdf（Delignat-Lavaud等，NDSS，2014年2月）。

② Perspectives Project，http://perspectives-project.org（检索于2014年5月27日）。

③ Convergence，http://convergence.io（检索于2014年5月27日）。

公钥钉扎解决了当前PKI生态系统里面最大的弱点，也就是任何CA都可以在未经域名拥有者同意的情况下给任意域名签发证书。有了钉扎之后，网站的拥有者可以选择（钉）一个或者多个他们信任的CA，创造他们自己单独的、比全球生态系统小很多的可信生态系统。公钥钉扎现在可以通过Chrome的专有机制实现，而HTTP公钥钉扎（public key pinning for HTTP）标准也在制定中。

- DANE

  DNSSEC是在DNS的基础上扩展了完整性检查的一组新协议。有了DNSSEC，域名可以与一组密钥关联起来，这一组密钥用于给对应DNS区进行签名。DANE是DNSSEC和TLS验证之间的桥梁。虽然DANE也可以用来做钉子，但是它最令人感兴趣的能力是完全绕过公共CA，如果你信任DNS，就可以用它做TLS验证。

- 主权密钥

  主权密钥（Sovereign Keys）提案[①]在现存的安全体系（包括CA和DNSSEC）中增加了更多的安全保障。主要的思路是域名可以声明使用主权密钥，这会记录在公开的、可验证的日志里面。声明之后，该域名的证书只能使用这个主权密钥进行签发才有效。它的问题是没有相关条款规定如果主权密钥泄露了我们该怎么办，这样就让该提案有很大的风险。主权密钥在2011年被提出，但是还处在构想阶段。

- MECAI

  CA是在一种公证人的概念上运行的体系，而互信CA基础设施（mutually endorsing CA infrastructure，MECAI）[②]则是这种公证人的一个变种。服务器处理所有的工作，然后将最新的证人发送给客户端。这大部分的过程都发生在幕后以便增强隐私和提高性能。MECAI在2011年被提出，但是还处在构想阶段。

- 证书透明度

  证书透明度（certificate transparency，CT）[③]是一套审计和监控公开证书的框架。CA将他们签发的每一张证书都提交到公开证书日志里面，然后获得一个此次已提交的加密证明。任何人都可以监控CA签发的每一张新证书。例如域名拥有者可以监控每一张签发了他们域名的证书。这个想法如果实现的话，那么虚假证书就可以快速地被发现。提交证明可以通过不同的方法交付给客户端（最好是在证书本身内部），这样就能用来确认该证书已经被公开了。从2015年2月开始，Chrome要求所有2015年之后签发的EV证书都有CT。[④]Chrome有一份白名单用来对那些在2015年1月份之前签发的EV证书进行识别，如果能证明该试点项目成功，Chrome的开发者倾向于最后所有的证书都需要有CT。

- TACK

---

① The Sovereign Keys Project，https://www.eff.org/sovereign-keys（EFF，检索于2014年5月27日）。

② Mutually Endorsing CA Infrastructure version 2，http://kuix.de/mecai/（Kai Engert，2012年2月24日）。

③ Certificate Transparency，http://www.certificate-transparency.org（Google，检索于2014年5月27日）。

④ Extended Validation in Chrome，http://www.certificate-transparency.org/ev-ct-plan（Ben Laurie，检索于2015年3月16日）。

- *证书密钥可信保证*（trust assurances for certificate keys，TACK）[1]是钉扎的一个变种，用来钉住服务器提供的签名密钥。引入一个长期的签名密钥意味着要做更多的工作，但是可以独立于整个CA体系。这个提案的不同之处在于，它支持所有使用TLS进行保护的协议，而不仅仅是HTTP。TACK于2012年提出，作者提供了一些主流平台的概念验证，不过直到撰写本书之时，还没有任何客户端中提供官方支持。

这些提案有希望得到实现吗？在2011年当大部分提案刚出来的时候，确实有很大的一股动力试图改变现状。不过在之后实现的时候，人们才认识到问题的难度。我们很容易设计在大多数情况下运转良好的系统，但是当遇到边界问题的时候，大部分系统都无法很好地工作。

基于公证人的提案所面临的问题是，浏览器的API才刚刚取得进展。这些提案致力于解决本地攻击的问题，但是遇到了太多的警告。通过依赖多个外部系统来获得信任，这些提案会增加决策难度（例如，如果不同公证人之间存在不一致，或者系统中来了一个*流氓分子*，rogue element），而且还引入了性能、可用性以及运行成本等多个问题。大型站点经常会给同一个域名部署多个证书，特别是从不同的地理位置观察时。这会导致的一个假象是：不止一个公证人是正确的。

钉扎的提案看起来比较有希望。有了钉扎之后，站点所有人就可以选择相信谁，这样就可以解决大量针对当前系统的攻击。Google在2011年部署了钉扎，使得DigiNotar CA被入侵的事情曝光。他们的专有钉扎机制在这时候还侦测到几起别的问题。希望钉扎功能可以在不久的将来通过一种标准化的机制对每个人都可用。

DANE是唯一从实质上改变我们信任机制的提案，但它的成功取决于操作系统和浏览器对DNSSEC的支持。浏览器供应商对这事暂时没什么激情，不过操作系统可能会最终实现。对于低风险的业务，DANE是一个很好的方案，甚至可以完全替代DV证书。另外一方面，高风险的业务对DNS的集中信任可能是一个潜在的问题；关键的问题在于无法避免政府的各种影响。现在支持DANE的还很少，不过随着DNSSEC的发展肯定也会越来越多的。

考虑到Google的影响力，CT是非常有可能成功的，不过需要经过几年时间，等有了足够多的部署，就会完全显现出效果。

总的来说，我们有两个方向可以并行地去做，它们会形成一个不同安全等级的多层系统。第一个方向是改进当前系统。例如Mozilla通过它的根证书库的影响力对CA们进行施压，以保证他们不会出现问题。事实上，CA承受了非常大的压力，导致在2012年CAB论坛的重组，并且确定了*Baseline Requirements*。从2010年开始增加的监控和审计行为帮助发现了很多小问题（现在大部分都解决了），而且继续对整个系统进行检查。最终，CT可能通过所有的公开证书存储库来实现公共信任的完全透明化。

第二个方向是使高风险网站可以选择更高的安全性。互联网PKI在实践中遇到的最大问题可能就是我们希望用一个系统解决所有人的问题。实际情况是，有很多的业务只需要简单的安全（低成本、低付出），一小部分业务希望能够有足够的安全而且愿意为其进行改造。新的技术（例如钉扎、HTTP严格传输安全、内容安全策略以及强制OCSP stapling）可以带来更高级别的安全。

---

[1] TACK，http://tack.io/（检索于2014年5月27日）。

第 4 章

攻击PKI

目前运行的公开密钥基础架构有一个天生的缺陷：所有的CA都可以在不经域名所有者同意的情况下给任意域名签发证书。很难想象这个已经使用了20多年的系统，依靠的竟然是每个人（数百家公司和上万人）去做正确的事。

现在已经有了几个可以利用的攻击方式。大多数案例中的验证流程都是攻击的目标。如果你能够让CA相信你是域名的合法拥有者，他们就会给你签发证书。其他情况下，攻击目标就是CA自身的安全体系。如果CA被入侵了，攻击者就可以给任意网站生成证书。一些已经真相大白的案例显示，某些CA签发了二级证书，这些二级证书泄露之后签发了大量的网站证书。

本章记录了从2001年第一个热点大事件开始到2013年年末为止，PKI最引人关注的事件和攻击。

4.1 VeriSign 签发的 Microsoft 代码签名证书

2001年1月，VeriSign被某个声称代表Microsoft的人欺骗，给他签发了两张代码签名证书。要实现这种攻击手段，攻击者需要建立假的身份，使得VerSign的一个或多个员工相信申请是经过授权的，并且需要为每个证书支付400美元的费用。换句话说，这需要攻击者拥有非常深厚的系统知识、技能以及决策能力。这次的问题在几周之后的日常审计中被发现。公众在3月末，也就是Microsoft发布了安全补丁之后知晓了这次事件。

这些虚假证书在操作系统上无法提供任何级别的信任，由它们进行代码签名的软件在运行的时候也会出现警告。因此，它们仍被认为对所有Windows系统的用户有危险；但因为签名信息用的是“Microsoft Corporation”，按常理来说，大部分用户在遇到有其签名的软件时都会同意安装。用Microsoft自己的话说就是：[1]

> 使用这些证书签名的程序无法自动运行或者绕过任何正常的安全限制。然而，在启动程序时出现的警告对话框会提示这些软件是由Microsoft进行数字签名的。很明显，这会误导很多用户去运行该程序。

发现这个错误之后，VeriSign就立刻吊销了这些证书，但这还不足以保护用户，因为这些虚

[1] Erroneous VeriSign-Issued Digital Certificates Pose Spoofing Hazard，https://technet.microsoft.com/library/security/ms01-017（Microsoft Security Bulletin MS01-017，2001年3月22日）。

假证书没有包含任何吊销信息。因此，在2001年3月末，Microsoft不得不发布一个紧急的软件升级将这些虚假证书加入黑名单，并且向用户解释如何找到它们。[①]这在随后催生了关于Microsoft Windows系统对证书吊销实现的在线讨论。[②]在那段时间里，Microsoft知识库里面有一篇文章介绍了如何从系统里移除可信的证书机构。[③]

4.2 Thawte 签发的 login.live.com

2008年夏天，安全研究人员Mike Zusman绕过Thawte的证书验证流程，获得了一张login.love.com的证书。login.love.com是Microsoft的单点登录验证中心，有几百万用户。

Mike的利用有两个前提：第一，Thawte使用邮箱作为域名所有权的验证；第二，Microsoft允许所有人注册@live.com邮箱地址。最常用的邮箱地址别名（例如，主机管理员或者Web管理员）都是被保留或者已经注册的，但是巧合的是Thawte允许的域名验证邮箱范围较大。Thawte接受用于验证的一个邮箱是sslcertificates@live.com，而用户可以注册获得这个邮箱。一旦Mike获得了该邮箱的访问权，获得证书就是迟早的事了。

虽然Mike在2008年8月份公开了这个问题，[④]但在年底才公布CA的名称。[⑤]他在第二年的DEFCON 17论坛上发表演讲，公布了漏洞利用细节。

七年之后的2015年，Microsoft又发生同样的事情，不过这次有问题的是live.fr域名。[⑥]

4.3 StartCom 违规（2008）

2008年10月19日，Mike Zusman通过利用StartCom网站的漏洞，设法绕过了StartCom的域名验证。[⑦]该漏洞是用来控制证书签发的，他可以利用该漏洞通过任意域名的所有权验证（StartCom提供两步验证流程：第一步需要证明你对该域名拥有所有权，第二步才是申请证书）。利用这一发现，Mike申请并获得了未经授权域名的两张证书。

他的攻击很快就被发现了，主要是因为他尝试继续获取授权，并且申请paypal.com和verisign.com的证书。原来，StartCom还有第二层的控制机制，那就是高危网站黑名单。这种深度

---

① 如何识别错误地颁发的VeriSign代码签名证书，http://support.microsoft.com/zh-cn/kb/293817（Microsoft，检索于2014年7月3日）。

② Microsoft, VeriSign, and Certificate Revocation，http://www.amug.org/~glguerin/opinion/revocation.html（Gregory L. Guerin，2001年4月20日）。

③ 如何从受信的根存储区中删除根证书，https://support.microsoft.com/zh-cn/kb/293819/（Microsoft，检索于2014年7月3日）。

④ DNS vuln + SSL cert = FAIL，http://intrepidusgroup.com/insight/2008/07/dns-vuln-ssl-cert-fail/（Intrepidus Group博客，2008年7月30日）。

⑤ Mike关于Thawte的推文，https://twitter.com/schmoilito/statuses/1089348859（2008年12月31日）。

⑥ A Finnish man created this simple email account - and received Microsoft’s security certificate，http://www.tivi.fi/Kaikki_uutiset/2015-03-18/A-Finnish-man-created-this-simple-email-account---and-received-Microsofts-security-certificate-3217662.html（Tivi，2015年3月18日）。

⑦ Nobody is perfect，http://schmoil.blogspot.com/2009/01/nobody-is-perfect.html（Mike Zusman，2009年1月1日）。

防御机制发现了Mike的行为，然后在几分钟之内吊销了所有非法签发的证书。

StartCom发布了一份详细的报告记录此次发生的攻击和事件。[①]Mike在DEFCON 17论坛上讨论了该事件更多的细节。[②]

4.4　CertStar（Comodo）签发的 Mozilla 证书

Mike Zusman攻击StartCom之后没几天，StartCom的首席执行官兼首席运营官Eddy Nigg就发现了另外一个CA存在相似的问题。[③]有一些垃圾推广邮件会误导他使用别的CA更新证书，[④]通过追踪这些邮件留下来的痕迹，Eddy Nigg发现了CertStar。它是Comodo在丹麦的合作伙伴，可以在没有进行域名所有权验证的情况下签发任何域名的证书。Eddy开始获得了startcom.org的证书，然后是mozilla.org。毫无疑问，一张伪造的Mozilla高危域名证书引起了轩然大波，人们在mozilla.dev.tech.crypto邮件列表里立刻对此事开始了讨论。[⑤]

核查了111张由CertStar签发的证书之后，Comodo吊销了11张（另外还有Eddy Nigg申请的两张）无法通过有效性验证的证书，并且宣称没有理由怀疑它们是伪造的。[⑥]

4.5　伪造的 RapidSSL CA 证书

2008年，由Alex Sotirov和Marc Stevens带领的一组研究人员完成了对互联网PKI体系的一次华丽攻击，验证了原本只处于理论上的攻击手段：他们设法获得了一张伪造的CA证书，可以用来给世界上任意一个网站签发证书。[⑦]

为了好好欣赏一下这次攻击，你需要理解之前针对MD5的整个攻击历史。你会发现这次是对MD5的最后一击，而针对MD5的攻击最早可以追溯到2004年的某个时间点，在那一年其实已经从理论上宣布MD5是可以攻破的。换句话说，这次攻击是持续努力的结果。

Marc Stevens和他团队的其他研究人员在2006年公布了不同特征的碰撞证书的相关工作，在2007年继续改进基于前缀选择的碰撞技术。他们可以使用自己的（私有）证书颁发机构，在他们能够完全控制的模拟环境里面生成碰撞证书。然而，在现实生活中，有诸多限制可以防止

① Full Disclosure，https://blog.startcom.org/?p=161（Eddy Nigg，2009年1月3日）。

② Criminal charges are not pursued: Hacking PKI（Mike Zusman，DEFCON，2009年7月17日）：幻灯片（http://defcon.org/images/defcon-17/dc-17-presentations/defcon-17-zusman-hacking_pki.pdf）和视频（http://www. youtube.com/watch?v=XCjoJePnRBY）。

③ (Un)trusted Certificates，https://blog.startcom.org/?p=145（Eddy Nigg，2008年12月23日）。

④ SSL Certificate for Mozilla.com Issued Without Validation，http://www.sslshopper.com/article-ssl-certificate-for-mozilla.com-issued-without-validation.html（SSL Shopper，2008年12月23日)。

⑤ Unbelievable!，https://groups.google.com/group/mozilla.dev.tech.crypto/browse_thread/thread/9c0cc829204487bf（mozilla.dev.tech.crypto，2008年12月22日）。

⑥ Re: Unbelievable!，https://groups.google.com/group/mozilla.dev.tech.crypto/msg/5f3824e5c2c55923（Robin Alden，2008年12月25日）。

⑦ MD5 considered harmful today，http://www.win.tue.nl/hashclash/rogue-ca/（Sotirov，2008年12月30日）。

这一利用。

MD5和PKI攻击时间表

- 1991年：Ronald Rivest设计了MD5，用来代替MD4。
- 1991~1996年：MD5开始流行起来，并且在很多应用中部署。与此同时，MD5暴露出一些缺陷的早期迹象[①]，让研究人员开始建议新的应用程序使用更加安全的散列函数。[②]
- 2004年：王小云等人[③]证明了完全碰撞。人们开始认为MD5被完全攻破了，但是方法太复杂，在实际中还无法实施。
- 2005年：Lenstra, Wang和de Weger证明了现实中的碰撞[④]，展示了两张完全不一样的证书，但却拥有同样的MD5散列值，即相同的签名。两张证书的RSA密钥空间是不一样的，但是其余信息都是相同的（例如证书上的身份信息）。
- 2006年：Stevens, Lenstra和de Weger提出一项新的技术[⑤]，该技术最开始被称为目标碰撞，之后改名为前缀选择碰撞。它可以创建两张身份信息不同，但是MD5散列值一样的证书。有了实际有效的攻击案例，MD5已经完全不安全了。
- 2008年：虽然MD5被认为不安全已经有十年之久，甚至在2006年还有一次实际而且有效的攻击证明，但是有一些证书机构还在继续使用它签发新的证书。由Sotirov和Stevens带领的一组研究人员通过使用MD5碰撞来完成针对PKI的攻击，并获得了一张“伪造”的CA证书，可以用它来生成任何网站的有效证书。[⑥]
- 2012年：发现了一种非常复杂的恶意软件火焰病毒（也被称作Flamer或者Skywiper）影响着中东的网络。[⑦]该恶意软件被认为是由政府赞助的，并且之后发现它针对Microsoft的CA证书采用了MD5碰撞的方式来攻击Windows Update的代码签名机制。在分析了相关证据之后，Marc Stevens推测该攻击使用了之前完全不知道的攻击手段。[⑧]没有人知道火焰病毒已经出现了多久，但是一般认为它是在最近的2~5年内变得活跃的。

---

① Collisions for the compression function of MD5，https://www.cosic.esat.kuleuven.be/publications/article-143.pdf（B. den Boer和A. Bosselaers，*Advances in Cryptology*，1993）。

② Cryptanalysis of MD5 Compress，http://cseweb.ucsd.edu/%7Ebsy/dobbertin.ps（H. Dobbertin，1996年5月）。

③ Collisions for hash functions MD4, MD5, HAVAL-128, and RIPEMD，https://eprint.iacr.org/2004/199.pdf（王小云等，2004）。

④ Colliding X.509 Certificates based on MD5-collisions，http://www.win.tue.nl/~bdeweger/CollidingCertificates/（Lenstra、Wang和de Weger，2005年3月1日）。

⑤ Colliding X.509 Certificates for Different Identities，http://www.win.tue.nl/hashclash/TargetCollidingCertificates/（Stevens、Lenstra和de Weger，2006年10月23日）。

⑥ MD5 considered harmful today，http://www.win.tue.nl/hashclash/rogue-ca/（Sotirov等，2008年12月30日）。

⑦ What is Flame?，http://www.kaspersky.com/flame（卡巴斯基实验室）。

⑧ CWI cryptanalyst discovers new cryptographic attack variant in Flame spy malware，http://www.cwi.nl/news/2012/cwi-cryptanalist-discovers-new-cryptographic-attack-variant-in-flame-spy-malware（CWI，2012年6月7日）。

4.5.1 前缀选择碰撞攻击

攻击者的目的是创建两个相同MD5签名的文档。大多数数字签名技术是对数据的散列值进行签名（不是直接对数据本身进行签名）。如果你可以创建两个拥有相同MD5散列值的文档，那么一个文档的签名对另外一个文档也是有效的。你需要做的全部事情就是将其中一个文档发送给可信机构，让他们对其进行签名，随后将签名复制到另外一份文档（伪造的版本）。

如果要用到证书领域，还有另外一个问题：你不能直接发送自己的证书给CA进行签名。相反，你会发送一些信息给他们（例如，域名和你的公钥），然后CA会生成证书。这种限制措施是有意义的，但是依旧可以被绕过。

碰撞攻击可以使用两个专门构建的碰撞块来完成，这些碰撞块用来操作散列算法，目的是让两个不同的输入有同样的状态。考虑到两个输入（一个是合法的文档，另一份是伪造的），就散列算法而言，碰撞块会破坏它们的不同。这意味着两件事情：(1) 你必须提前知道合法文档的前缀（也就是前缀选择名称的来源）；(2) 你需要能够将其中一个碰撞块插入进去。

在实际中是不可能将碰撞块插入到最末尾的，这也是为什么最终碰撞的文件需要有相同的后缀。换句话说，当你碰撞成功之后，肯定不希望文件里面有任何的变化，否则散列值又变了。

4.5.2 创建碰撞证书

要在现实中使用前缀选择技术，那么我们的攻击就必须受制于那些我们想要伪造的文档的结构要求，而且要遵从文档创建和数字签名的流程。

一般的数字签名会有以下这些限制。

(1) 证书由证书机构签发，签发的信息来自提交的CSR文件。

(2) 证书的总体结构需要按照X.509 v3规范，攻击者无法改变结构，但是可以预测。

(3) 证书的某些信息会直接从CSR文件复制过来，攻击者可以完全控制这一块。更严重的是证书里面的公钥都是从CSR里面逐字复制的。这个公钥可以设计为随机的，也就是说通过精心设计、看起来像随机数的碰撞块不会引起任何警报。

(4) 证书机构还会将一些别的信息加入到证书里面。攻击者也许可以影响其中一部分（例如，证书过期时间），但是总体而言，他们这里能做的事情是预测内容会是什么。

从上面这段信息可以明确碰撞前缀必须在公钥之前包含所有的证书字段（也就是存储碰撞块的地方）。因为碰撞块的内容依赖前缀，整个前缀必须提前知道，这样才能创建碰撞数据，并且随后发送给证书机构。大部分前缀里面的证书字段都是公开的（例如，颁发者信息可以从另外一个由同一个CA签发出来的证书中获得），或者由攻击者在CSR文件里面提供（例如，公用名）。然而，有两个字段是由CA控制并且无法实现得知的：证书序列号以及过期时间。暂时，我们假设攻击者能够预测这两个字段的内容；之后我们会看看如何实现。

我们还得解决如何处理公钥之后的部分（后缀）。原来，这部分由多个X.509扩展组成，都是可以提前知晓的。通过适当的对齐（MD5按照块操作数据），两张证书的后缀可以很容易变得一样。

因此，攻击过程如下所示。

(1) 确定CA生成的证书前缀是什么，以及确定一些CSR字段的内容。

(2) 给伪造的证书刻意构建一个前缀。

(3) 确定后缀。

(4) 从前面三步的数据构建碰撞块。

(5) 构建CSR文件并提交给证书机构。

(6) 通过将伪造的前缀、第二部分的碰撞块以及后缀合并起来构建一张伪造的证书，其中的签名信息从真正的证书中获取。

> **注意**
> 第二部分的碰撞块和后缀必须是伪造证书的一部分，这样攻击才能生效，但是又必须通过某种方式隐藏起来，这样证书在使用的时候才不会出现问题。这次RapidSSL的攻击中，一个称为**肿块**（tumor）的代码被放置到不重要的X.509 v3注释扩展中，在处理过程中会被忽略掉。有经验的人可以发现这些异常，但是事实上没有人会在这个级别上检测证书。

4.5.3　预测前缀

现在我们回过头来讨论研究人员是如何设法预测那两个字段的内容(过期时间和序列号)的，因为每一张证书里面这两个字段都是不一样的。事实证明，这里面有运气的成分，再加上证书机构的"帮助"。我们来看看这是如何做的。

- RapidSSL的证书签发过程是完全自动化的，而且一般都会在CSR文件提交之后的6秒完成证书的生成。这就意味着可以比较精确地预测证书的过期时间，误差在1秒左右，这已经足够了。
- RapidSSL不是采用随机序列号（随机序列号是最佳实践），而是采用简单的计数器，每签发一张证书就加一。这意味着如果你快速获取连续的两张证书，就可以预测第二张证书的序列号。

在那个时候还有6家CA会签发基于MD5的证书，但是RapidSSL因为这两个原因，再加上缺乏其他防御措施，[①]最终让整个攻击得以实现（如图4-1所示）。然而，有一个麻烦事是使用该小组特别定制的、由200台PlayStation 3控制台组成的计算集群，还需要大概一天的时间才能生成一个碰撞。因此，他们不仅需要选择好提交CSR的准确时机，还要预测证书会被签发的序列号。

① 很显然，PKI是一项有非常多技巧的业务，这也是为什么从密码学角度来看会有各种各样的最佳实践以及深度防御措施来作为最后的兜底。X.509 v3扩展那些可以为别的证书进行签名的证书设计了基本限制字段扩展，并且将CA标志位的值设置为`true`。该扩展还有一个参数叫作`pathlen`，可以用来限制子CA的深度。如果RapidSSL的CA证书的`pathlen`被设置为`0`（意味着不允许签发出二级 CA），这张伪造CA证书将毫无用处。

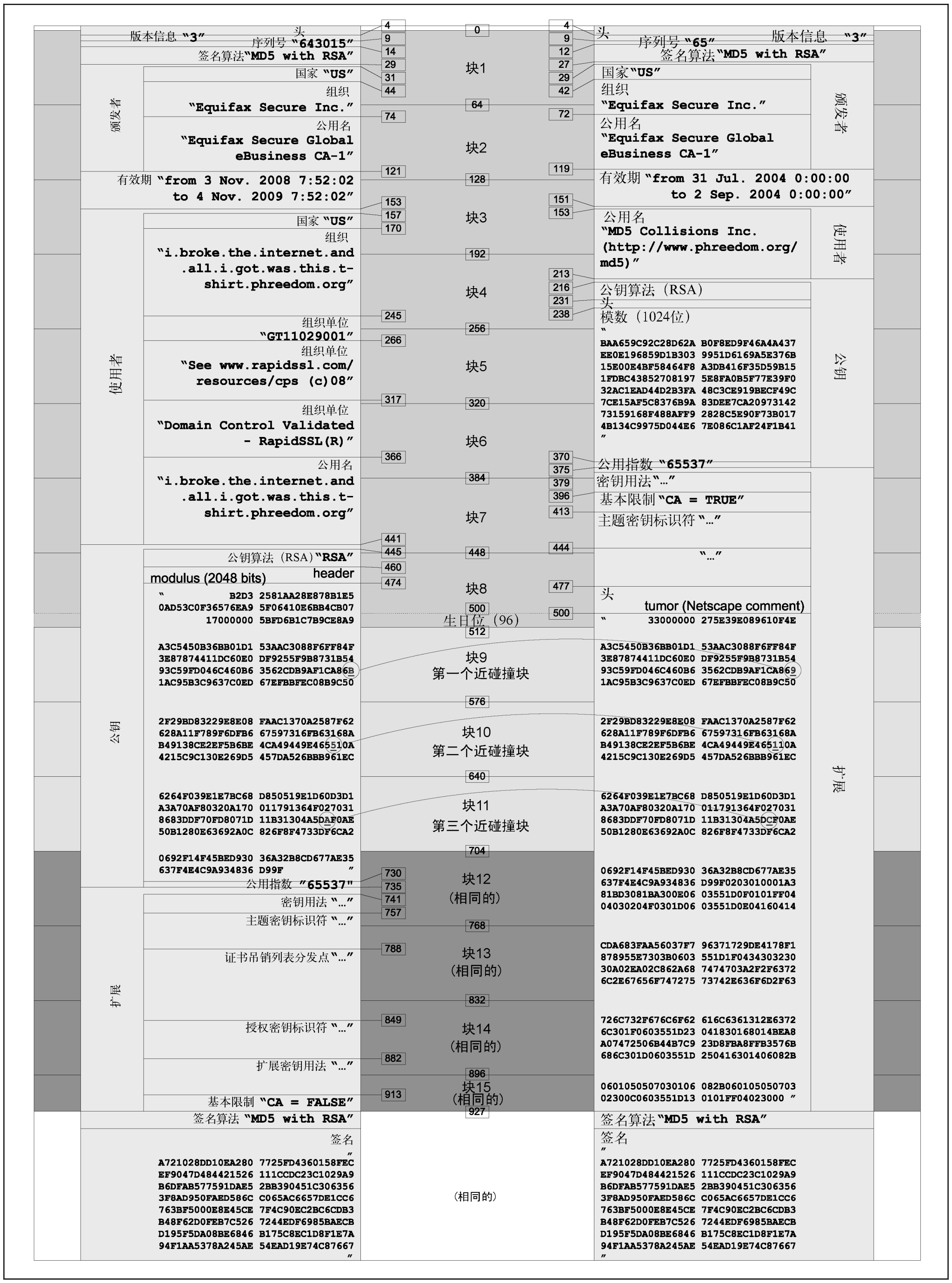


图4-1 真实证书（左边）和碰撞的RapidSSL证书（右边）的对比（来源：Benne de Weger）

他们的方法是在周日晚上展开攻击，这是CA最闲的时间段。他们会在周五获得序列号计数器的值，目标是提交一份CSR文件之后序列号依旧低于1000。当要进行攻击的时候，他们会通过申请新的证书推动计数器上涨，目标是尽可能地接近1000。每个周末他们都有足够的时间尝试三次。在失败了三周之后，他们在第四周终于成功了。

4.5.4 接下来发生的事

在计划这次攻击之前，研究人员就采取了措施，尽可能地降低潜在的附加风险。例如，给伪造证书构造的过期时间是过去的某个时间点，意味着即便对应的私钥泄露了，这张证书也是无用的。在公布此次攻击之前已经事先告知了负责浏览器可信库的重要机构（例如，Microsoft、Mozilla等），让他们可以先将这张伪造CA证书加入到黑名单里。他们还事先警告了RapidSSL，[①]这促使他们加速迁移到了SHA-1。他们在公开宣布之后的几个小时之内就升级到SHA-256。[②]在安全问题全部修复之后，研究人员才公布前缀选择碰撞技术的所有细节。

结果，这次攻击只花了657美元的证书费用，[③]当然研究者有权限访问一个200台PS3组成的计算机集群。EC2上同样的CPU能力需要花费20 000美元。在发布这次攻击公告的时候，研究人员估计通过一些优化手段，可以在使用2000美元的情况下，用一天的时间重复这次攻击。

4.6 Comodo 代理商违规

2011年，从3月份的另一次Comodo泄露事件开始，一系列的事件开始浮出水面。第一次的攻击发生在3月15日，Comodo的其中一个注册机构（registration authority，RA）被"完全入侵"（Robin Alden和Comodo首席技术官的原话），导致给7个网站一共签发了9张证书，[④]相关站点如下。

- addons.mozilla.org
- global trustee
- google.com
- login.live.com
- login.skype.com
- login.yahoo.com
- mail.google.com

显然，除了不知道生成名为global trustee证书的意图之外，其他所有的证书都是给那些每天有几百万用户访问的站点使用。幸运的是，这次攻击很快就被发现了，所有伪造的证书在几小时

---

① Verisign and responsible disclosure，http://www.phreedom.org/blog/2009/verisign-and-responsible-disclosure/（Alexander Sotirov，2009年1月6日）。

② This morning's MD5 attack-resolved，http://www.symantec.com/connect/blogs/mornings-md5-attack-resolved（Tim Callan，2008年12月30日）。

③ 虽然他们申请了大量的证书，但是其中的大部分都是补签的证书，而RapidSSL允许免费补签证书。

④ Comodo Report of Incident，https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html（Comodo，2011年3月22日）。

之内都被吊销了。事实上甚至不清楚攻击者是否拿到了这些证书。Comodo在他们的OCSP响应服务器上只看到有查询Yahoo证书的记录（而且只有两次），其他证书则没有出现过任何的OCSP请求记录。[①]

第二天，Comodo开始通知其他相关方，同时开始修复问题。[②]虽然Comodo没有公开被入侵RA的身份，不过之后攻击者声称被入侵的是意大利Instant SSL公司。在3月22日，Comodo、Mozilla和Microsoft等公司一起公开了此次攻击。

一个有趣的事情是，有一些人提前几天就从Chrome源代码的一些线索中发现了此次攻击（源代码是公开的）。Jacob Appelbaum在他的Tor博客中写到了他的发现。[③]

之后的3月26日，Comodo又公布了两起代理商被入侵的事件，不过其中一个后来被确认是假报告。另一份报告是真实的，不过该代理商并未签发任何伪造证书。显然，在3月15日出事之后，新采取的安全措施有效地阻止攻击者继续签发新的证书。[④]

同样在3月26日，攻击者开始与公众进行沟通，[⑤]我们也在这时知道了ComodoHacker（攻击者给自己取的名称），结果之后其实背后还有更深的故事：长达数月的行动，涉及众多CA，发生了相当多事件。你会在之后读到更多关于他的信息。

到了5月，Comodo又一次被卷入舆论中心，因为他们的一个代理商ComodoBR的网站上被发现存在SQL注入漏洞。[⑥]攻击者利用该漏洞获得用户的隐私数据（包括证书签发请求），不过没有引起其他与PKI有关的后果。

最后，这一连串的事件暴露出，要运营一个大型的合作伙伴网络，采用监护人的管理方式是很难实施的，特别是在PKI这样一个复杂的生态系统里面。Comodo宣称在2008年的事件之后，只有9%的合作伙伴还保留着签发证书的能力，但即便如此还是晚了。在2011年的第一次事件之后，除非过了Comodo的验证，否则所有的代理商都不能够直接签发证书。

更重要的是，这些事件说明Comodo（也许还有其他的CA）没有维护一个现实的威胁模型。Robin Alden在关于mozilla.dev.security.policy的一张贴子中承认了此事（我在重点部分加粗了）：

> 我们之前建立的威胁模型，很多RA可能无法很好地完成，甚至试图避免做有效性检查，这本是他们的职责（不过这次出事的RA都没出现过这两种情况），**不过我们没有充分考虑到这种新的威胁模型，RA成为了被攻击的重点，最终导致完全入侵。**

---

① 严格来说，这并不能完全说明证书没有被使用，因为主动中间人攻击者可以抑制来自受害者的OCSP流量。

② Bug 642395: Deal with bogus certs issued by Comodo partner，https://bugzilla.mozilla.org/show_bug.cgi?id=642395（报道于2011年3月17日）。

③ Detecting Certificate Authority compromises and web browser collusion，https://blog.torproject.org/blog/detectingcertificate-authority-compromises-and-web-browser-collusion（Jacob Appelbaum，2011年3月22日）。

④ RE: Web Browsers and Comodo Announce A Successful Certificate Authority Attack, Perhaps From Iran，https://groups.google.com/ forum/#!msg/mozilla.dev.security.policy/zgKmHOTIxn8/5NNYcgPNqlgJ（Robin Alden，2011年3月29日）。

⑤ A message from Comodo Hacker，http://pastebin.com/74KXCaEZ（ComodoHacker，2011年3月26日）。

⑥ New hack on Comodo reseller exposes private data，http://www.theregister.co.uk/2011/05/24/comodo_reseller_hacked/（The Register，2011年5月24日）。

4.7　StartCom 违规（2011）

2011年夏天，StartCom又一次成了被攻击的对象。按照推测，攻击者与之前攻击Comodo的人是同一个。[①]这次事件发生在6月15日，此事发生之后，StartCom停止签发新证书的时间长达一周。下面是其网站上的消息：

> 由于6月15日发生了针对我们系统的攻击以及出现安全泄露，所以数字证书签发以及相关的服务暂时停止。请等待进一步的通知。订阅人和有效证书的拥有者不会受到影响。网站的访客以及依赖于有效证书的第三方不会受到影响。我们对此带来的不便深表歉意，感谢您的理解。

显然，虽然没有任何伪造的证书被签发，攻击者依旧有可能已经获取了某些敏感数据的访问权限，甚至可能非常接近公司最重要的根密钥[②]，竟然没有造成严重的长久损害。官方在这之后也没有发布这次事件的报告，只能通过Eddy Nigg的一篇博客获得有关这次事件的更多内容。[③]

4.8　DigiNotar

DigiNotar是荷兰的一家公司，主营业务是给大众签发证书，同时处理荷兰政府电子政务项目相关的PKI部分，项目名称叫作PKIoverheid（其中的overheid表示荷兰政府）。2011年，DigiNotar成为第一家完全被入侵的CA，由此伪造的证书在现实生活中被使用，而且可能产生了非常严重的中间人攻击。毫无疑问，DigiNotar的根证书完全被吊销了，这家公司也逐渐失去了业务，并且在2011年9月份自愿宣布破产。

4.8.1　公众的发现

8月27日，伊朗的一位Gmail用户报告访问电子邮箱账户时好时坏，此时真相开始逐渐浮出水面。[④]根据记录的证词，每天会出现30~60分钟的“不可用”，表现为在访问的时候弹出“不正常证书”的安全警告，导致无法访问。后来发现用户描述的不可用是因为Chrome检测到出现了中间人攻击，并利用他们自己独有的公钥钉扎技术阻止用户继续访问。

接下来的几天，我们获悉，之前报告的问题只是冰山一角，背后是一次从未听说过的、影响面非常大的攻击，预计受影响的IP地址达到300 000个。事实上，所有这些IP地址都是在伊朗，所有被拦截的证书都是由DigiNotar签发出来的。但是这种情况怎么可能发生？

---

① Another status update message，http://pastebin.com/85WV10EL（ComodoHacker，2011年9月6日）。

② Response to some comments，http://pastebin.com/GkKUhu35（ComodoHacker，2011年9月7日）。

③ Cyber War，https://blog.startcom.org/?p=229（Eddy Nigg，2011年9月9日）。

④ Is This MITM Attack to Gmail’s SSL ?，https://productforums.google.com/forum/#!msg/gmail/3J3r2JqFNTw/oHHZLJeed-HMJ（alibo，2011年8月27日）。

4.8.2 一个证书颁发机构的倒下

DigiNotar这次严重的安全事件甚至影响到荷兰政府整个数字体系，所以荷兰政府立刻控制了DigiNotar并且聘请外部安全咨询公司Fox-IT进行调查。Fox-IT在一周之后的9月5日发布了第一次报告。[①]下面是报告中与此次事件关联最大的部分内容：

> 最关键的服务器上面存在的恶意软件在正常情况下是可以被杀毒软件检测到的。关键组件的分离并没有起到作用或者运行不适当。有很强的迹象表明，CA的服务器虽然在物理上存放在非常安全的环境中，但是依旧能够通过内部网络进行访问。
>
> 整个网络完全被入侵了。所有CA服务器都是一个Windows域的成员，导致有可能使用同一个用户名密码组合就可以登录所有服务器。密码强度也不够，非常容易受到暴力破解。
>
> 安装在对外公开Web服务器上的软件已经过期了，而且没有打任何补丁。
>
> 在被调查的服务器上没有任何的杀毒软件保护。
>
> 有一个运行着的入侵防御系统，但是不清楚为什么它在那个时候没有阻止外部Web服务器的攻击。不存在安全中心网络的任何日志记录。

整个完整的报告在一年之后的2012年8月份公开，共100页，详细记录了CA被入侵的所有细节。[②]从报告中我们知晓最开始的攻击发生在6月17日，一个对外公开的Web服务器运行了有漏洞的内容管理系统并且被入侵了。从那时起直到7月1日，攻击者才入侵到最安全的网络节点，也就是存放根的相关设备的地方。这个网络虽然没有与互联网直接连接，但是攻击者可以通过一些不那么重要的系统作为跳板连通过去。

第一批的128张虚假证书是在7月10日被批量签发的，大约是在攻击者获得CA服务器访问权限后的一周。之后还有几次批量签发动作，一共给53个独立的域名签发了至少531张证书。由于被入侵的缺口太大，实际签发的证书数量无法得知；日志也被篡改了，还有很多之后在外面发现的证书无法在对应的数据库里面找到任何记录。

你可以在表4-1里看到一连串的名称，包含了一些访问量很高的网站、证书颁发机构以及政府机构。

表4-1 DigiNotar攻击者签发的证书包含的公用名

..com	*.*.org	*.10million.org (2)
*.android.com	*.aol.com	*.azadegi.com (2)
*.balatarin.com (3)	*.comodo.com (3)	*.digicert.com (2)
*.globalsign.com (7)	*.google.com (26)	*.JanamFadayeRahbar.com
*.logmein.com	*.microsoft.com (3)	*.mossad.gov.il (2)

① DigiNotar public report version 1，https://www.rijksoverheid.nl/ministeries/ministerie-van-binnenlandse-zaken-en-koninkrijksrelaties/documenten/rapporten/2011/09/05/diginotar-public-report-version-1（Fox-IT，2011年9月5日）。

② Black Tulip Update，https://www.rijksoverheid.nl/documenten/rapporten/2012/08/13/black-tulip-update（荷兰政府，2012年8月13日）。

（续）

*.mozilla.org	*.RamzShekaneBozorg.com	*.SahebeDonyayeDigital.com
*.skype.com (22)	*.startssl.com	*.thawte.com (6)
*.torproject.org (14)	*.walla.co.il (2)	*.windowsupdate.com (3)
*.wordpress.com (14)	addons.mozilla.org (17)	azadegi.com (16)
Comodo Root CA (20)	CyberTrust Root CA (20)	DigiCert Root CA (21)
Equifax Root CA (40)	friends.walla.co.il (8)	GlobalSign Root CA (20)
login.live.com (17)	login.yahoo.com (19)	my.screenname.aol.com
secure.logmein.com (17)	Thawte Root CA (45)	twitter.com (18)
VeriSign Root CA (21)	wordpress.com (12)	www.10million.org (8)
www.balatarin.com (16)	www.cia.gov (25)	www.cybertrust.com
www.Equifax.com	www.facebook.com (14)	www.globalsign.com
www.google.com (12)	www.hamdami.com	www.mossad.gov.il (5)
www.sis.gov.uk (10)	www.update.microsoft.com (4)	

其中一些证书对应的站点并不出名，但是主要用来传输不同的消息。证书里面有很多地方出现了如表4-2所示的这些短语。

表4-2 证书里面内置的消息（不能确定翻译是否准确）

原始消息	翻　译
Daneshmande Bi nazir	Peerless scientist/杰出的科学家
Hameye Ramzaro Mishkanam	Will break all cyphers/破解所有的密码
Janam Fadaye Rahbar	I will sacrifice my life for my leader/我会为我的领导牺牲生命
Ramz Shekane Bozorg	Great cryptanalyst/伟大的密码学家
Sahebe Donyaye	Possessor of the world (God)/世界的掌控者（神）
Sare Toro Ham Mishkanam	I will break Tor too/我还会破解Tor
Sarbaze Gomnam	Unknown soldier/不知名的战士

报告中还透露出DigiNotar在7月19日就发现了这次入侵，并且在外部咨询公司的帮助下，在7月底清理了他们的系统。不幸的是，破坏措施已经实施。大概是担心这次事件可能导致的影响，DigiNotar偷偷地吊销了一小批伪造的证书（他们所知道的那一部分证书），但是却没有通知任何人。

4.8.3 中间人攻击

考虑到入侵的严重程度，即使DigiNotar立刻对外宣布被入侵，可能也无法挽救DigiNotar的命运，不过至少可以立刻阻止攻击者继续使用伪造的证书。我们之所以可以知晓，是因为伪造的证书里面带上了OCSP的信息，调查人员可以通过检测DigiNotar的OCSP响应程序的日志来跟踪证书的部署情况。①

① 如果TLS客户端遇到了包含OCSP信息的证书，它会连上指定的OCSP服务器来确定证书是否被吊销了。这种方式并不完全准确，因为中间人攻击者可以阻塞所有到OCSP服务器的流量。浏览器遇到OCSP通信失败的时候会直接忽略掉。

最开始，在证书生成之后，从日志里面发现了几次请求：可能是攻击者正在测试。第一次大量部署的迹象从8月4日开始，一直持续增加到8月29日，这一天也是浏览器吊销DigiNotar根证书、同时杀死所有伪造证书的日期。也许是因为攻击手段有限（最可能使用的是DNS缓存污染手段），[①]从被攻击的用户口中得知攻击不是连续的，而是某一时间突然爆发，也可能是因为无法一次性处理大流量的访问（如图4-2所示）。



图4-2　DigiNotar OCSP在2011年8月的活动（来源：Fox-IT）

此外，攻击者可能只在意收集Gmail的密码。假设他们的容量有限，一旦从某个来源IP请求中看到了密码信息，就会切换到拦截别的IP。有了密码缓存之后，他们就可以在闲暇时间直接访问Gmail来滥用这些账号（因为人们很少修改密码）。

总之，一共有来自298 140个独立IP地址的654 311个OCSP请求，检测了伪造Google证书的吊销状态。IP地址有大约95%处于伊朗境内，剩下的查证为全球的Tor退出节点、代理或者虚拟私有网络。

4.8.4　ComodoHacker 宣布负责

ComodoHacker于9月5日在他的Pastebin账号中宣布对这次DigiNotar入侵事件负责。[②]之后他又发了三篇文章，并附上了一个二进制的calc.exe文件，该文件使用其中一个伪造证书进行签名。该证据强有力地证明他参与到了这次事件中。文章还包括了此次攻击的一些细节内容，这与官方

① DNS缓存污染是一种通过利用DNS协议的弱点以及实现，来发起针对DNS基础设施的攻击。使用一些聪明的技巧带上大量的包，有可能使得DNS缓存服务器对域名的代理从原本真正的拥有者手中切换到攻击者上。成功之后，攻击者就可以在它的攻击空间里面决定哪些域名返回哪些IP地址。一次成功的攻击可以影响所有使用同一个缓存DNS服务器的用户。在伊朗发生的这次DigiNotar中间人攻击中，一些用户反馈称，将他们的DNS设置从ISP的服务器改为别的服务器（例如，Google的），攻击就停止了。

② Striking Back...，http://pastebin.com/1AxH30em（ComodoHacker，2011年9月5日）。

报告提供的信息完全吻合（官方报道是在很久之后才对外公开的）。

> 我是如何绕过DigiNotar互联网服务器之后的6层网络的；如何找到密码；如何在打满补丁和最近更新的系统里面获得系统权限；如何绕过它们的nCipher NetHSM、硬件密钥、RSA证书管理员以及内部第6层的“证书专用网络”（这个专用网络与整个互联网完全隔绝）；如何在防火墙只允许80和443端口而不允许反向和直接的VNC链接的情况下获取完整的远程桌面连接；还有很多很多……

然而依旧不能确定ComodoHacker是否真正参与了在伊朗的这次攻击。虽然他很高兴地宣称对此次的CA入侵负责，但是ComodoHacker说自己并没有参与到中间人攻击。他发的第二篇DigiNotar文章包含了下面的内容：

> 我就是一个人，不要再说我在伊朗组建了一支军队。如果有人在伊朗用了我生成的证书，这和我没有关系。

在之后的一篇文章中，他重复了此类声明：

> 就我一个黑客，我就是将一些证书分享给伊朗的一些人，仅此而已……我就一个人，你们知道就好。

谁是ComodoHacker？

ComodoHacker于2011年第一次出现在公众面前，留下了PKI相关的一些痕迹，以及一连串的针对多个证书机构的攻击。第一次批量的攻击发生在2011年的3月，Comodo的多个合作伙伴被入侵了。伪造的证书被签发出来，不过很快就被发现，这也阻止了这些伪造证书被利用。

StartCom是在6月份被攻击的，攻击者获得了一部分的成功，不过根据双方所说，没有签发出任何伪造证书。StartCom将签发证书业务停了一段时间，但是没有提供任何关于此次事件的进一步细节。

之后就是DigiNotar受到攻击，这导致整个DigiNotar证书机构被完全入侵，甚至动摇了整个PKI生态体系。

ComodoHacker在一条消息里提及对GlobalSign的攻击是一次成功的入侵，所以GlobalSign停止了一段时间的证书签发业务并且启动调查。他们之后发现是对外Web服务器被入侵了，而不是CA体系。[①]唯一出问题的是www.globalsign.com域名的私钥被拿到了。

在Comodo事件之后，攻击者开始通过ComodoHacker账号在Pastebin上与公众开始对话，[②]总共留下了10条信息。在DigiNotar事件之后，他在Twitter上以ich sun的名字发表了一些短暂的推文，并且操作着ichsunx2账号。[③]虽然他最开始出现的时候很享受聚光灯下的感觉，甚至

---

① September 2011 Security Incident Report，https://web.archive.org/web/20140202142309/https://www.globalsign.com/company/press/121311-security-incident-report.html（GlobalSign，2011年12月13日）。

② ComodoHacker’s Pastebin，http://pastebin.com/u/ComodoHacker（检索于2014年8月7日）。

③ ich sun的Twitter，https://twitter.com/ichsunx2（检索于2014年8月7日）。

做了一些专访，不过在2011年9月最后一次在Twitter上与公众联系之后，他就消失了。

4.9 DigiCert Sdn. Bhd.

2011年11月，马来西亚的证书机构DigiCert Sdn. Bhd.被发现签发了危险的不安全证书。这个公司与美国另外一家更出名的公司Digicert Inc.没有关系。DigiCert Sdn. Bhd.是相继与CyberTrust（Verizon）和Entrust有过合作关系的一家中间证书颁发机构。一共发现22张证书存在弱密钥问题，而且缺少一些关键的属性。

- 不安全的512位密钥

  只有512位长度的密钥使用暴力破解可以非常容易地重构出来。[①]有了密钥之后，那些恶意的团伙就可以假扮成正常的网站，而浏览器不会弹出任何警告。

- 缺少用途限制

  所有证书必须在扩展密钥用法（extended key usage，EKU）字段里面带上用途限制。虽然在与Entrust的合同里面限制了DigiCert Sdn. Bhd.只允许签发服务器站点的证书，但是因为这些伪造证书上面缺少了用途限制，那么就可以用于其他目的，例如代码签名。

- 缺少吊销信息

  这22张证书都没有包含吊销信息。意味着即便发现了无效的证书也没有任何方式可以可靠地吊销它们。

事实证明，这个问题是在一个公开的密钥被人用暴力破解并用于签发恶意软件才被发现的。[②]发现这个问题之后，Entrust吊销了这个中间证书[③]并且通知了所有的浏览器厂商。在一周之内，Entrust和CyberTrust都吊销了各自的中间证书，Mozilla在他们的博客上发文通知公众，[④]浏览器厂商发布更新将这些中间证书和所有已知的弱服务器证书加入到黑名单里面。在这之后，DigiCert, Inc.不得不向他们的用户解释他们与DigiCert Sdn. Bhd.是两家不同的公司。[⑤]

4.10 火焰病毒

2012年5月，安全研究人员开始分析一种新型的恶意软件，它主要在中东肆虐。这个恶意软件

① 不是暴力破解所有可能的数字。更高效的方式是使用正数因素分解方法，例如，General number field sieve（GNFS）。

② Bug #698753: Entrust SubCA: 512-bit key issuance and other CPS violations; malware in the wild，https://bugzilla.mozilla.org/show_bug.cgi?id=698753（Bugzilla@Mozilla，2011年11月1日）。

③ Entrust Bulletin on Certificates Issued with Weak 512-bit RSA Keys by Digicert Malaysia，https://web.archive.org/web/20150629012844/http://www.entrust.net/advisories/malaysia.htm（Entrust，检索于2014年7月3日）。

④ Revoking Trust in DigiCert Sdn. Bhd Intermediate Certificate Authority，https://blog.mozilla.org/security/2011/11/03/revoking-trust-in-digicert-sdn-bhd-intermediate-certificate-authority/（Mozilla Security Blog，2011年11月3日）。

⑤ DigiCert, Inc. Of No Relation to Recent “Digi” Insecure Certificates，https://www.digicert.com/news/2011-11-1- breaches-and-similar-names.htm（DigiCert, Inc.，2011年11月1日）。

叫作火焰病毒（Flame，也被称为Flamer或者Skywiper），是那时候最先进的病毒：超过20 MB的大小和超过20个攻击模块（一般的恶意软件包括网络嗅探、打开麦克风和文件获取等），而且使用轻量级关系型数据库（SQLite）和脚本语言（Lua）等组件。它以这种形式隐藏了相当长一段时间（意味着非常低的感染率或者无法被察觉到，很明显这不是一般水平的人能开发出来的）。

总的来说，火焰病毒在大约1000个系统中被发现，看起来是一种目标性非常明确的攻击（如图4-3所示）。伊朗CERT于2012年5月份发表了一篇关于火焰病毒的新闻稿。[1]不久之后，火焰病毒的创造者发出了自杀命令，让所有的火焰病毒将自身删除。不过之后还是发现了很多恶意软件存留，并且捕获和分析了几个命令和控制服务器的实例。[2]

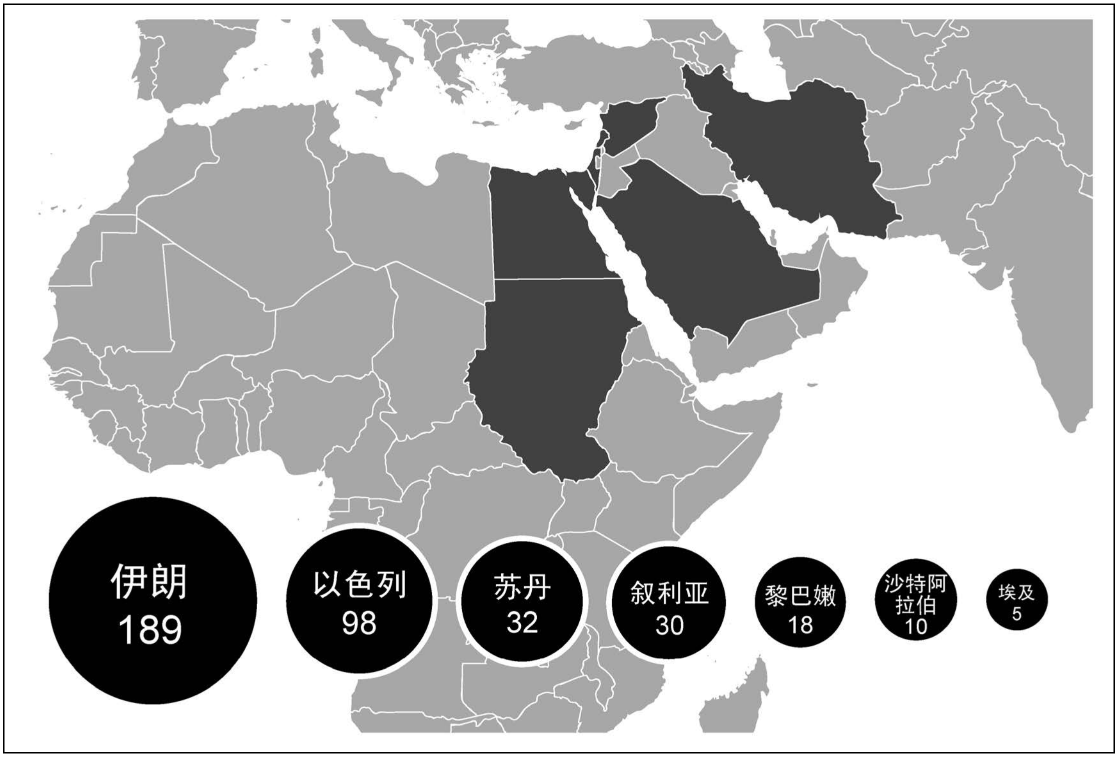


图4-3　火焰病毒活动图（来源：卡巴斯基实验室）

4.10.1　火焰病毒对抗 Windows 更新

之后发生的事情震惊了所有人。据报道，火焰病毒一个功能是攻击Windows的更新机制，可

① Identification of a New Targeted Cyber-Attack，https://web.archive.org/web/20140331222159/http://www.certcc.ir/index.php?name=news& file=article&sid=1894（MAHER，2012年5月28日）。

② Flame/Skywiper/ Flamer reports，http://www.crysys.hu/targeted-attacks.html（CrySyS Lab，2012年5月31日）。

以传播到本地网络里面安装的任意Windows系统。更让人惊讶的部分是，火焰病毒使用密码学作为攻击手段来完成这次攻击。[①]更重要的是，这种特定的攻击手段在之前是不为人知的。

一旦处于同一个本地网络，破坏Windows的更新机制就非常容易了。Internet Explorer支持Web代理自动发现机制（web proxy autodiscovery，WPAD），是一种可以让程序在本地网络自动找到HTTP代理服务器的协议。[②]有本地网络访问权限的攻击者能够以代理的身份进行广播，从而获得受害者的HTTP(S)的流量。火焰病毒正是这样做的，而且包含一个简单的Web服务器去假装成Windows更新服务器，来广播带有恶意代码的可用“更新”。[③]

Windows更新系统没有使用TLS（在我的桌面系统上进行了简单测试，结果显示所有的更新流量都是明文的），但是Microsoft确实会在更新的时候用到代码签名，也就是说应该无法创造出来自于Microsoft的二进制文件。这个事件里面最奇怪的就是火焰病毒竟然能够以Microsoft的身份给所有二进制文件进行签名。

4.10.2 火焰病毒对抗 Windows 终端服务

当Microsoft开始讨论遭到火焰病毒攻击的弱点的时候，一个更深藏的问题暴露了出来。为了运营终端服务许可（terminal services licensing），每台终端服务装置在激活的时候会收到特殊的二级CA证书。然后使用这个二级CA创建最终用户许可。Microsoft在设计系统的时候出现了以下几个严重的错误。

(1) 主终端服务CA证书（用来给每个单独的客户签发二级CA）与Windows更新CA一样，都是由同一个可信的根签发出来的。

(2) 父终端服务CA可以用于许可，以及出于一些无法解释的原因，用于代码签名。

(3) 二级CA证书没有用途限制，意味着对它们的约束限制与它们的父证书是一样的。

这些意味着每一个终端服务器用户都拥有一个不受限制的二级CA，这个CA可以用来给Windows更新的二进制文件进行签名，不需要任何的黑客行为。

幸运的是，这些证书只能用于对付安装Windows XP的机器。二级CA证书包含了一个独有的X.509扩展，名为Hydra，被标记为关键扩展。[④]

Windows XP的代码在进行证书校验的时候忽略了这个关键扩展，但是从Windows Vista（于2007年1月30日在全球发布）以及之后的Windows版本开始，都能够理解这个关键性的扩展并且很好地处理它。也就是说火焰病毒的作者必须找到一种没有Hydra扩展的证书。

---

① Analyzing the MD5 collision in Flame，http://blog.trailofbits.com/2012/06/11/analyzing-the-md5-collision-in-flame/（Alex Sotirov，2012年6月11日）。

② Web Proxy Autodiscovery Protocol，https://en.wikipedia.org/wiki/Web_Proxy_Autodiscovery_Protocol（维基百科，检索于2014年7月3日）。

③ Snack Attack: Analyzing Flame’s Replication Pattern，https://threatpost.com/snack-attack-analyzing-flames-replication-pattern-060712/76660/（Alexander Gostev，2012年6月7日）。

④ 在PKI里面，如果某个扩展被标记为关键扩展，客户端只有能够识别这个扩展才能完成证书链的验证，否则验证就会失败。这个特性后面的想法是关键扩展也许包含一些必要的信息，识别这个特性是强校验的前提。

4.10.3　火焰病毒对抗 MD5

Microsoft犯下的另外一个严重错误是在设计终端服务器许可协议的时候给证书用的是MD5签名。其他的错误（4.10.2节中讨论过的）相对来说没这么容易，并且需要对PKI的了解非常深入才能发现。但是在Microsoft设计系统的时候，MD5已经被广泛认为是不安全的。到了2008年，MD5的不安全性已经有了非常有效的证明——在对RapidSSL的攻击中生成了一张伪造的证书。从长远角度来看，Microsoft那时候根本就不应该在他们的证书程序里面允许MD5证书，但却将它们用在了终端服务许可上。

如果你阅读了前面讲的RapidSSL攻击以及伪造CA证书的生成过程，也许可以猜测到后面发生的事情：火焰病毒利用针对MD5的前缀选择碰撞攻击来生成一张伪造的CA证书。这次攻击原理上与之前介绍的RapidSSL攻击是一样的。下面是我们所知道的一些事情。

(1) 因为使用了不安全的MD5签名，所以可以对系统进行密码学相关的攻击。

(2) 证书签发过程是自动的，而且攻击者可以控制时间。除了证书有效期和序列号之外所有字段都可以事先知晓。

(3) 证书有效期只需要秒级精度，可以被预测。

(4) 序列号不像RapidSSL那样逐步加一，但也可以被预测（启动之后的毫秒数加上两个固定的字节和一个序列证书号），不过需要毫秒级精度。

对毫秒级别的精度要求可能让整个事情变得困难很多，因为需要一个非常好的网络连接来降低抖动。如果可以访问高性能计算集群将会加速碰撞搜索并提高准确性。我们无法得知需要尝试多少次（如果Microsoft很好地记录了许可行为，那么也许他们能够知道），不过攻击者最终显然是成功了。

Marc Stevens，那位前面发表前缀选择碰撞攻击技术的主要人物，分析了这张伪造证书作出如下判断：[①]

> 火焰病毒使用了前缀选择碰撞攻击……火焰病毒使用生日搜索算法，带上4个近似碰撞块获得一个碰撞。
>
> 这些碰撞位对正常证书来说是藏在原始证书的RSA模数中的，而对于伪造的证书则是在issuerUniqueID字段里面。用我自己的工具可以获取原始证书的近似碰撞块（该工具无法提供）并且在第一个近似碰撞块之前获得了链值。有了这些信息我就可以重构4个不同的路径。这些不同的路径展示出一条不同的前缀选择碰撞攻击，可以用于构建一条还未记录在案的全新的不同路径算法。

无论是谁设计了火焰病毒并对Microsoft实施了攻击，他们一定有自己非常厉害的硬件、一个非常厉害的开发团队以及世界级的密码学家。

① Microsoft Sub-CA used in malware signing，http://lists.randombit.net/pipermail/cryptography/2012-June/002992.html（Marc Stevens，2012年6月12日）。

反密码分析

针对用于签名的散列函数的碰撞攻击是一个真正的威胁。虽然MD5的问题离我们已经很远了，但是现在被广泛使用的SHA1也被认为是不安全的。在理想的世界中我们现在本应该已经停止使用它了。但是在现实中，它还会存在很多年，因为我们还需要面对复杂的生态体系和大量的不作为。

为了应对这些问题，Marc Stevens发明了反密码分析，[①]它用于寻找被成功碰撞攻击的证书的痕迹，正如研究论文的摘要所描述的那样：

> 我们采用一种反密码分析作为一种新的范例，来加强弱加密基元以便对抗密码分析攻击。将原有的弱基元进行重新设计来对抗密码分析，这种技术不可避免地会导致向后兼容的问题。相反，反密码分析通过利用密码分析攻击过程中不可避免的痕迹，可以检测和阻止密码分析攻击，同时又能保证向后的兼容性。

4.11 TURKTRUST

2012年12月，多亏了Chrome浏览器支持的公钥钉扎技术，Google发现了另外一起严重的PKI问题。钉扎允许用户客户端对一些站点进行检测，但只有特定授权的CA可以签发该站点的证书。Chrome里面有一个小的、硬编码的站点列表，不过这些站点都是世界上访问量非常大的网站。[②]

2012年12月，一个Chrome用户遇到了一张与内置的硬编码列表不相符的证书，然后浏览器将整个有问题的证书链传输回Google。看到证书链之后，Google发现这张伪造证书一直链到土耳其的证书机构TURKTRUST。[③]

这些无效的二级证书立刻被所有方面吊销了。TURKTRUST在几天之后就发布了详细的报告，并且持续更新报告内容。[④]我们获悉TURKTRUST在2011年8月份的两个系统设施的切换过程中犯了一个错误，导致那天签发的两张证书被标记为CA证书。这个问题连续15个月都没有被检测出来，在这段时间内这两张证书都被当作普通的服务器证书在使用。

在2012年12月的某个时间点，EGO安装了一台拥有MIMT功能的防火墙，EGO是其中一个拥有误签发的二级CA证书的组织。承包商将这个证书导入到防火墙，并开始按需生成伪造的网站证书来进行中间人监控。在这个过程中生成了一张Google的证书并且投入使用，随后被Google检测到。

---

① Counter-cryptanalysis，https://marc-stevens.nl/research/papers/C13-S.pdf（Marc Stevens，CRYPTO 2013）。

② 我会在第10章中对公钥钉扎进行讨论。

③ Enhancing digital certificate security，http://googleonlinesecurity.blogspot.com/2013/01/enhancing-digital-certificate-security.html（Google Online Security Blog，2013年1月3日）。

④ Public Announcements，https://web.archive.org/web/20130926134541/http://turktrust.com.tr/en/kamuoyu-aciklamasi-en.2.html（TURKTRUST，2013年1月7日）。

无法确定承包商是否知道这张有问题的证书其实是CA证书。如果你在排查一台MIMT设备的问题，而且对PKI不熟悉，就可能会尝试导入任何一张你手边的有效证书。

浏览器根证书库运营人员接受了TRUKTRUST将此次事件定位为管理错误。暂时不存在任何证据表明存在对该CA的攻击，在EGO自己的网络之外也没有发现任何伪造的证书。Mozilla要求TURKTRUST进行一次非例行审计，同时Google和Opera决定停止识别TURKTRUST的EV证书。

4.12　ANSSI

2013年12月，Google宣布吊销ANSSI（国际计算机安全局，法国的网络信息安全局）签发的二级证书。几天之后，ANSSI证书机构被限制只能签发法国地区域名的证书（.fr是法国地区使用的主要顶级域名）。[①]

这次吊销的原因是因为发现该二级CA证书被用于该机构网络里面运行的透明窃听（中间人）设备。结果签发了多个不同域名的证书，其中一些域名属于Google。Chrome的钉扎技术又一次检测出滥用PKI的情况。

Mozilla[②]和Microsoft[③]也禁用了被滥用的CA证书。当局发表了简短的声明，将问题归咎于人为错误。没有任何证据表明伪造的证书在French Treasury的网络之外存在。[④]

通常情况下，接下来会对`mozilla.dev.security.policy`进行讨论。[⑤]

随着该事件的细节被进一步披露之后，很多其他问题（比如ANSSI是如何使用CA证书的）也暴露出来。例如，他们的很多证书都没有包含任何吊销信息。他们的证书吊销列表存在不正常的活动，数以千计的证书突然出现在之前完全是空的列表中。不确定这个事情是如何发现的。ANSSI坦诚他们至少要到2015年12月份才能遵从*Baseline Requirements*的规范，这比Mozilla给的期限要晚两年。[⑥]

---

① Further improving digital certificate security，http://googleonlinesecurity.blogspot.co.uk/2013/12/further-improving-digital-certificate.html（Google Online Security Blog，2013年12月7日）。

② Revoking Trust in one ANSSI Certificate，https://blog.mozilla.org/security/2013/12/09/revoking-trust-in-one-anssi-certificate/（Mozilla Security Blog，2013年12月9日）。

③ 不正确颁发的数字证书可导致欺骗，https://technet.microsoft.com/library/security/2916652（Microsoft Security Advisory 2916652，2013年12月9日）。

④ Revocation of an IGC/A branch，https://web.archive.org/web/20141202143116/http://www.ssi.gouv.fr/en/the-anssi/events/revocation-of-an-igc-a-branch-808.html（2013年12月7日）。

⑤ Revoking Trust in one ANSSI Certificate，https://groups.google.com/forum/#!topic/mozilla.dev.security.policy/uCZi6Ua-uPA（mozilla. dev.security.policy，2013年12月9日）。

⑥ Announcing Version 2.1 of Mozilla CA Certificate Policy，https://blog.mozilla.org/security/2013/02/15/announcing-version-2-1-of-mozilla-ca-certificate-policy/（Mozilla Security Blog，2013年2月15日）。

4.13 印度国家信息中心

2014年7月，Google检测到几张误签发他们域名的证书，然后一路跟踪到中间证书，发现属于印度国家信息中心（National Information Centre，NIC），而这张中间证书是由印度控制器证书颁发机构（Controller of Certifying Authorities，CCA）签发的。之后发现它的二级CA（NIC证书颁发机构或者说NICCA）被入侵了，伪造签发了很多Google和Yahoo的域名证书。这张有问题的中间证书被立刻吊销，NICCA也完全停止了证书签发。在Chrome里面，根证书CCA机构被限制只能签发几个印度（.in）的子域名证书。[①]

4.14 广泛存在的 SSL 窃听

虽然PKI有很多弱点，但是对整个生态体系最大的威胁在于广泛存在的SSL窃听，这些窃听是由本地安装的软件、雇主以及网络提供商实施的。虽然我们可能很少听说，但久而久之，这类窃听变得越来越普遍。那些被影响的人可能无法察觉出什么事情，也就不会上报这类事件。幸运的是，有人会偶尔上报这类事件，我们对这类问题的警觉性又会提高一些。

4.14.1 Gogo

2015年1月，Adrienne Porter Felt（Google Chrome安全团队的一名成员）上报空中互联网连接公司Gogo会窃听所有加密的信息，并且提供了一些带有合法网站域名的无效证书。[②]

Gogo没有办法生成有效的证书，也就是说用户必须点击忽略证书警告才能访问他们想访问的站点，因此这种证书是无效的；但是这无法改变一个事实就是，Gogo可能可以毫无限制地访问用户的敏感信息。

Adrienne的Twitter发言触动了整个社区的神经，收到了大量的回复，商业公司对它的用户实施的网络攻击在媒体上成为热点事件。Gogo之后发表声明，将这种行为归咎于需要控制飞机上网络带宽的使用情况。[③]虽然该公司也许真的认为窃听行为是有必要的，但是这种声明是站不住脚的。一段时间之后，Gogo完全停止了窃听行为。[④]

---

① Maintaining digital certificate security，http://googleonlinesecurity.blogspot.co.uk/2014/07/maintaining-digital-certificate-security.html（Google Online Security Blog，2014年7月8日）。

② hey @Gogo, why are you issuing *.google.com certificates on your planes?，https://twitter.com/__apf__/status/551083956326920192（Adrienne Porter Felt，2015年1月2日）。

③ Our Technology Statement from Gogo regarding our streaming video policy，https://web.archive.org/web/ 20150404014855/http://concourse.gogoair.com/technology/statement-gogo-regarding-streaming-video-policy（Gogo，2015年1月5日）。

④ Gogo no longer issuing fake Google SSL certificates，http://www.runwaygirlnetwork.com/2015/01/13/gogo-no-longer-issuing-fake-google-ssl-certificates/（Runway Girl Network，2015年1月13日）。

4.14.2　Superfish 和它的朋友们

就在一个月之后，也就是2015年的2月，Adrienne又发表了一项公开声明，那就是联想（Lenovo）在他们的一些系统上预装了广告插入软件Superfish（有些人认为是恶意软件）。[①]引起我们兴趣的是，这个软件会对所有用户流量进行劫持，包括那些安全加密的网站。为了避免出现证书警告，Superfish在用户毫不知情的情况下将一个有问题的“可信”根证书加入到操作系统的证书库里面。有了这个根证书之后，Superfish就将所有的浏览器流量引向本地的代理进程。然后这个进程取到网站返回的内容之后，就可以做任何它想要做的修改了。

先不说这是否存在道德问题（这个广告插入软件可以看到所有的流量，不管多么隐私或者敏感），他们使用的方法也是错误的。正确的劫持方式是对每个用户都需要生成唯一的根证书。Superfish在他们所有系统里面使用了同一个根证书，这意味着任何一个受到此类影响的用户都可以导出对应的根证书的私钥，然后用他攻击所有其他被影响的用户。不出所料，这个有问题的根证书在这次公开声明之后很快就被提取出来了。[②]（也许很久之前就已被导出，不过我们没有这类信息。）

还有一些别的问题：每个用户笔记本里面的代理软件相比浏览器而言，在TLS功能上比较弱；它仅仅支持TLS 1.1，而不是最新的TLS 1.2。同时还是有许多不安全的密码套件，极大地降低了用户的安全性。更糟糕的是，他无法正确地验证无效的证书。相当于可以合法地使用自签名证书进行中间人攻击。有句话还得再次强调一下：无论访问任何网站，带有Superfish的用户都是无法看到证书警告的。[③]

> **注意**
> 本地安装的根证书可以绕过像钉扎之类的严格安全手段，这是有意为之。这种方式可以允许企业进行SSL监听（也许是合法的）。这也是为什么Superfish甚至可以监听到Google的流量，虽然Google已经采取了非常多的安全措施。

Facebook的分析指出Superfish影响全球非常多的用户。在哈萨克斯坦，Superfish影响了Facebook中4.5%的连接数。[④]

联想最开始试图为他们的行为进行辩护，但最后还是妥协了，最终与Microsoft合作将Superfish和所有不该有的根证书从受影响的系统中移除。Microsoft发布的数据显示，他们仅仅几

① #Superfish Round-Up，https://noncombatant.org/2015/02/21/superfish-round-up/（Chris Palmer，2015年2月22日）。

② Extracting the SuperFish certificate，http://blog.erratasec.com/2015/02/extracting-superfish-certificate.html（Robert Graham，2015年2月19日）。

③ Komodia/Superfish SSL Validation is broken，https://blog.filippo.io/komodia-superfish-ssl-validation-is-broken/（Filippo Valsorda，2015年2月20日）。

④ Windows SSL Interception Gone Wild，https://www.facebook.com/notes/protect-the-graph/windows-ssl-interception-gone-wild/1570074729899339（Facebook，2015年2月20日）。

天就从大约250 000台系统中移除了Superfish。[①]

> **注意**
> 与之前一样，出现这类事情之后就会有很多线上测试出现。Filippo Valsorda发布了对Superfish、Komodia和PrivDog的测试。[②]Hanno Böck则对其他相似的产品进行了测试。[③]

更深入的调查显示Superfish是使用Komodia提供的SSL监听SDK构建的。从Komodia的站点来看：

> 我们的SSL劫持者SDK是一个全新的技术，它允许你访问那些使用SSL加密的数据，并实时进行SSL解密。劫持者使用Komodia的转向器平台来让你可以非常容易地访问、修改、跳转以及记录数据，而不出现任何浏览器证书错误提示。

Superfish不是唯一的SSL监听产品，也不是唯一具有安全缺陷的SSL监听产品。虽然Comodo的产品PrivDog（版本3.0.96.0）采用每个用户独立监听根的方式，但依旧无法正确地进行证书校验，使中间人攻击变得容易起来。[④]

Komodia、Superfish以及PrivDog是最先进入我们视野的，但还有很多相似的产品。随着安全人员开始重视起来，还有其他一些类似的产品浮出水面，让众人得以知晓。其中还有一些非常著名的安全产品。[⑤]

4.15 CNNIC

2015年3月，一家叫作中东通信系统（Mideast Communication Systems，MCS）的公司收到CNNIC（China Internet Network Information Center，中国互联网信息中心，是一家负责中国互联网的机构）的一张测试中间证书。MCS原本是希望在埃及市场提供证书以及相关服务。不幸的是，在测试过程中，中间证书被导入到一台可以进行透明SSL监听的设备里面，导致最终至少产生了一张误签发的证书。使用Chrome浏览器的一个工程师将此证书传回Google。[⑥]

① MSRT March: Superfish cleanup，http://blogs.technet.com/b/mmpc/archive/2015/03/10/msrt-march-superfish-cleanup.aspx（Microsoft Malware Protection Center，2015年3月10日）。

② Superfish, Komodia, PrivDog vulnerability test，https://filippo.io/Badfish/（Filippo Valsorda，检索于2015年3月22日）。

③ Check for bad certs from Komodia / Superfish，https://superfish.tlsfun.de（Hanno Böck，检索于2015年3月22日）。

④ Comodo ships Adware Privdog worse than Superfish，https://blog.hboeck.de/archives/865-Software-Privdog-worse-than-Superfish.html（Hanno Böck，2015年2月23日）。

⑤ The Risks of SSL Inspection，https://insights.sei.cmu.edu/cert/2015/03/the-risks-of-ssl-inspection.html（Will Dormann，2015年3月13日）。

⑥ Maintaining digital certificate security，http://googleonlinesecurity.blogspot.co.uk/2015/03/maintaining-digital-certificate-security.html（Google Online Security Blog，2015年3月23日）。

经过调查，Google和Mozilla决定吊销对CNNIC根证书的信任。[①]不过CNNIC提供了一份白名单证书，在吊销之前由其签发出来的有效证书还是可以继续使用的。虽然是MCS的行为导致了证书误签发，但是CNNIC仍被认为有过失，因为它给一家没有能力维护CA的组织签发了一张毫无限制的有效中间CA证书。[②]

① Distrusting New CNNIC Certificates，https://blog.mozilla.org/security/2015/04/02/distrusting-new-cnnic-certificates/（Mozilla Security Blog，2015年4月2日）。

② Consequences of mis-issuance under CNNIC，https://groups.google.com/forum/#!msg/mozilla.dev.security.policy/czwlDNbwHXM/Fj-LUvhVQYEJ（mozilla.dev.security.policy讨论主题，开始于2015年3月23日）。

第 5 章

HTTP和浏览器问题

在本章中，我们将讨论TLS和HTTP的关系。TLS设计用于保护TCP连接，但是当今的浏览器中有很多持续变化的东西存在。在很多场景下，许多问题是由于浏览器厂商致力于兼容老旧网站而导致的：他们担心破坏Web的兼容性。

5.1 sidejacking

sidejacking是网络应用会话劫持的一种特殊情况，在这种情况下，攻击者会从一条未加密的连接上获取会话令牌[①]。这种攻击在无线网络或者本地局域网中非常容易实施。如果一个网站没使用加密流量，攻击者只需要简单地观察网络流量并从中提取会话令牌即可。如果网站只是部分启用了加密，则可能会出现以下两种问题。

- 设计导致的会话泄露

  某些网站使用加密来保护账户密码，但是当身份验证结束后就切换回明文。这个方法对安全性有微小的增加，但是这只是将信息的泄露点由密码转移到了会话令牌上。从某种程度上看，会话令牌确实价值更小一些，因为它们只是在有限的时间内有效（假设会话管理功能是正确实现的），但是它们却十分容易被窃取并且很容易被主动攻击者滥用。

- 错误导致的会话泄露

  即使你很努力地将整个网站都使用加密，也很容易出现遗漏一两个资源还是明文访问的错误（如图5-1所示）。即使主页面加密保护起来，同一域名下的一个单独的明文资源也可能引起会话泄露[②]。这种问题叫作*混合内容*（mixed content）问题，我会在本章的后面讨论。

sidejacking可以针对任何类型的会话令牌进行攻击，因为攻击者可以完全获取受害者与服务器之间的通信。因此，这种攻击不仅可以窃取保存在Cookie中的会话令牌，也可以获取保存在URL中的令牌（路径或者请求参数）。一旦获取令牌，攻击者就可以使用这个数据以受害者的身份直接访问目标网站。

① 在Web应用中，当用户连接到网站的时候，一个新的会话就会被创建出来。每个会话都会被分配一个秘密令牌（也称为会话ID），主要是用来标明会话的所有权。如果攻击者找到了一个已经过身份验证的会话令牌，就可以在网站上以受害者的身份获取全部权限。

② 这主要是因为会话令牌通过Cookie传送，这个Cookie会存在于每个发向网站的请求中。

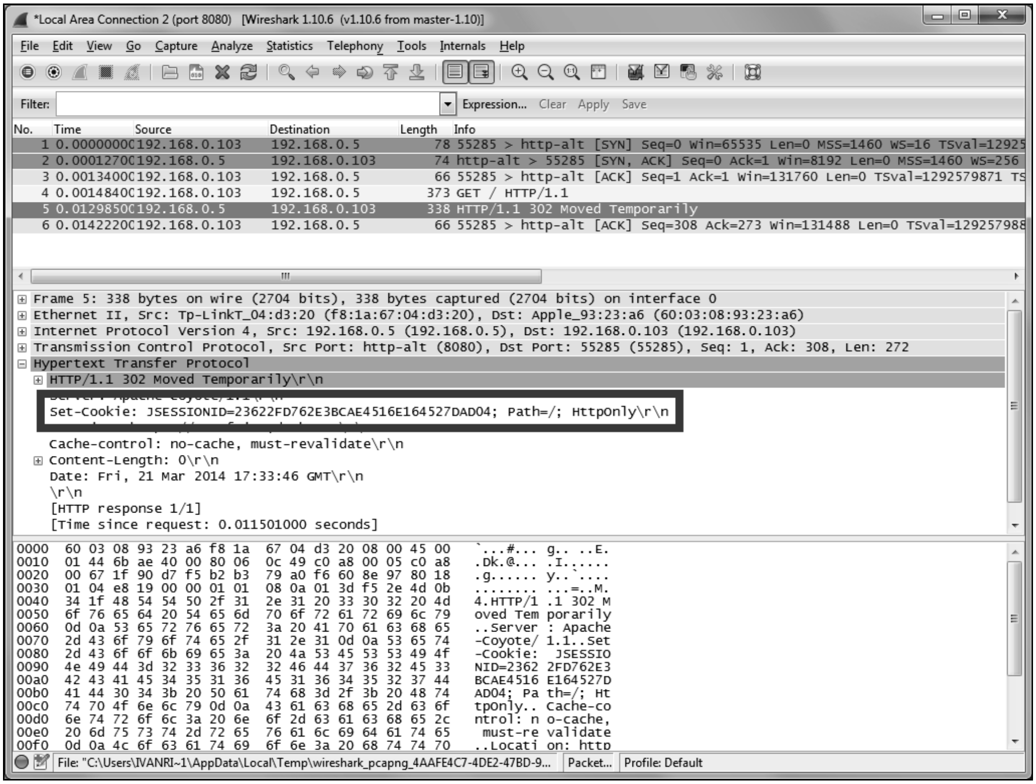


图5-1　在Wireshark中显示明文的会话Cookie

在安全社区中，sidejacking在2007年8月得到了人们更多的认识。当时，Robert Graham和David Maynor在美国黑帽子大会上讨论了这个话题，并公开了用于自动化攻击的Ferret和Hermit工具[①]。

几年之后，一个由Eric Butler开发的名为Firesheep[②]的Firefox插件引发了更大的轰动，因为该插件使得sidejacking攻击更容易实施。Firesheep变得非常出名，几个高价值的网站都切换到了全站加密。紧随Firesheep之后，名为BlackSheep[③]的检测工具和名为FireShepard[④]的反击工具相继开发出来。此外，还有一个叫作Idiocy[⑤]的工具用来自动对受攻击的账号发出警告。

Firesheep已经不再维护。目前类似的工具有CookieCadger[⑥]，这是一个由Matthew Sullivan开发的被动HTTP审计工具。

---

① SideJacking with Hamster，http://blog.erratasec.com/2007/08/sidejacking-with-hamster_05.html（Robert Graham，2017年8月5日）。

② Firesheep announcement，http://codebutler.com/firesheep/（Eric Butler，2010年10月24日）。

③ BlackSheep，https://www.zscaler.com/blacksheep.php（Zscaler，2014年7月15日）。

④ FireShepard，https://notendur.hi.is/gas15/FireShepherd/（Gunnar Atli Sigurdsson，检索于2014年7月15日）。

⑤ Idiocy，http://jonty.co.uk/idiocy（Jonty Wareing，检索于2014年7月15日）。

⑥ CookieCadger，https://www.cookiecadger.com/（Jonty Wareing，检索于2014年7月15日）。

5.2 Cookie 窃取

正如我们刚才讨论的那样，sidejacking对于全站加密的网站是无效的。在这种情况下，会话令牌会一直被隐藏到加密层之后。你也许认为这种完全部署的TLS意味着sidejacking真的毫无用处，但实际上并非如此。程序员常犯的一个错误是忘记对Cookie进行加密，当这种问题发生时，攻击者可以使用一种叫作Cookie窃取（cookie stealing）的技术来获得会话令牌，如图5-2所示。

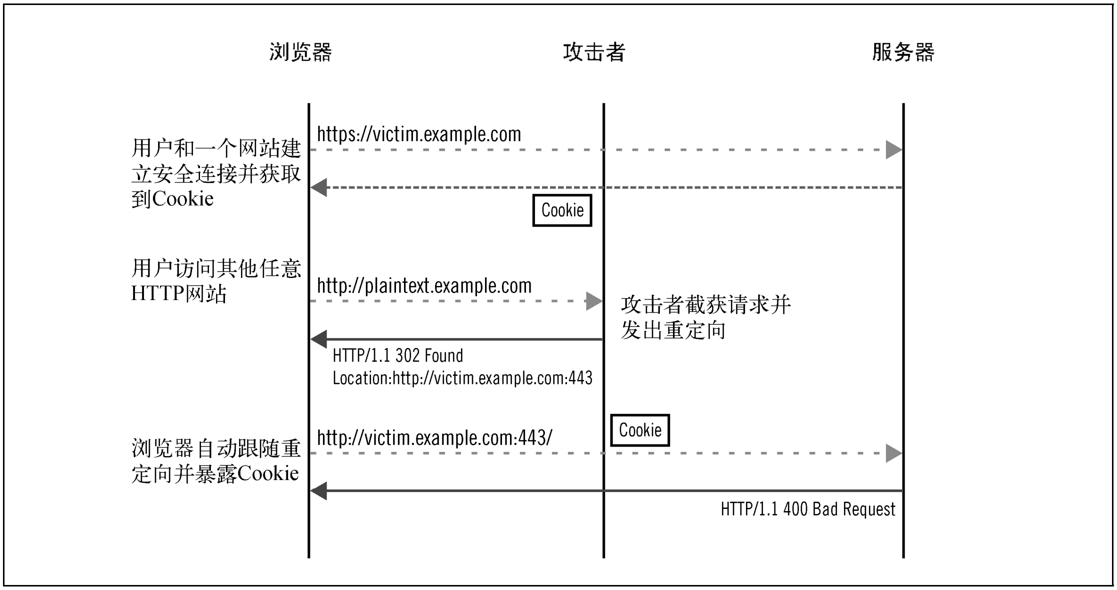


图5-2 使用中间人攻击窃取不安全Cookie

默认情况下，无论是80端口的不安全传输还是443端口的安全传输，Cookie都会被发送。在对网站部署TLS时，你也期望所有的Cookie都会同样安全，期望浏览器能正确处理。如果你不采取措施，起初看来可能没什么问题，因为用户都被完全保护着。但是这只有在浏览器不向80端口发送数据的时候才可以，如果攻击者可以找到一种方法让浏览器向80端口发送数据，Cookie就有可能被窃取。

理论上，这种攻击很简单：攻击者是一个可以观察到受害者网络流量的主动中间人攻击者。攻击者无法对目标网站的加密流量进行攻击，但是他可以等待受害者发出一个发往任意其他网站的非加密HTTP请求。在这种情况下，攻击者开始介入，劫持不安全连接，然后将受害者的不安全请求重定向到目标网站的80端口。因为任何网站都可以将请求重定向到其他网站，所以浏览器不会产生怀疑。

最终的结果就是受害者会向目标网站建立一个明文连接，其中包含所有的Cookie。针对典型的Web应用程序都不将Cookie标识成安全，攻击者就可以获取到受害者的会话令牌并继续进行会话劫持攻击。

这种攻击在目标网站即使不对80端口进行响应的情况下也可以实施，由于攻击者位于中间位置，因此他可以截获到任何端口的明文信息。

另外一种攻击者可能会使用的方法是将受害者重定向到相同的主机名和443端口（对于安全网站来说443端口是打开的），但以http://www.example.com:443这样的方式使之强制发送明文信息。虽然这个请求最终会失败，因为浏览器向加密端口发送了明文请求，但是由于请求中已经包含全部Cookie信息，因此攻击者也会成功。

Mike Perry是第一个将此问题公之于众的人，就在sidejacking公开之后不久。但是他发往BugTraq邮件列表的邮件[①]却没有引起足够的重视。他后来在DEFCON 16上发表了相关演讲[②]坚持自己的观点，同时发布了它的感念验证工具CookieMonster[③]。

5.3　Cookie 篡改

Cookie篡改攻击主要发生在攻击者无法直接获取到Cookie的场景下，在这种情况下，Cookie一般都得到了很好的保护。利用Cookie规范中的弱点，攻击者可以注入新的Cookie，或者覆盖、删除已有的Cookie。本节传达的主要信息是，Cookie的完整性并不能被完全保证，即使是在网站完全加密的情况下。

5.3.1　了解 HTTP Cookie

HTTP Cookie是一种用来在客户端保存少量数据的扩展机制。对于要设置的每个Cookie，服务器必须制定一对名称和值，以及描述其作用范围和生命周期的元数据。Cookie的创建是通过使用`Set-Cookie`响应头来实现的：

```
Set-Cookie: SID=31d4d96e407aad42; Domain=www.example.com; Path=/; Secure; HttpOnly
Set-Cookie: lang=en-US; Expires=Wed, 09 Jun 2021 10:18:14 GMT
```

客户端在所谓的cookie-jar中存储Cookie。对于每个HTTP事务，客户端在它们的cookie-jar中查找Cookie，并使用`Cookie` HTTP请求头将它们提交给服务器：

```
Cookie: SID=31d4d96e407aad42; lang=en-US
```

Cookie在最初发明出来的时候，缺少严格的定义，这种情况持续了很长时间。结果就是，具体的实施状况很不一致并且充满漏洞。就像你在本章中将看到的那样，很多漏洞都可以被利用来执行攻击。针对Cookie的良好的定义在2011年出现，就是在RFC 6265[④]中。

从安全的观点来看，Cookie的问题有两个方面：(1) 它们从一开始就没有被很好地设计，导致了安全漏洞产生；(2) 它们没有与当今浏览器使用的主流安全机制同步，这指的是同源策略

① Active Gmail“Sidejacking”- https is NOT ENOUGH，http://seclists.org/bugtraq/2007/Aug/70（Mike Perry，2007年8月5日）。

② HTTPS Cookie Stealing，http://fscked.org/talks/ActiveHTTPSCookieStealing.pdf（Mike Perry，2008年8月4日）。

③ CookieMonster，http://fscked.org/projects/cookiemonster（Mike Perry，检索于2014年7月15日）。

④ RFC 6265: HTTP State Management Mechanism，http://tools.ietf.org/html/rfc6265（A. Barth，2011年4月）。

（same-origin policy，SOP）。

- **宽松主机名范围**

  Cookie被设计成跨特定域名的不同主机名来使用，也可以跨不同协议和端口使用。一个为example.com使用的Cookie，可以在所有的子域名上生效（例如，www.example.com和secure.example.com）。同样，一个类似blog.example.com域名的Cookie默认只对blog.example.com有效（当`Domain`参数没有指定的时候），但是也可以显式地将其有效范围扩展到父域名。这样一来，恶意服务器可以将Cookie注入到安装在共享同一个域名的主机上的其他网站或应用。我把它们叫作*相关主机名*（related hostname）或者*相关网站*（related sites）。

  这个关于有效范围的宽松策略与SOP规则相冲突，SOP定义了一个严格匹配协议、主机名和端口的安全上下文。

- **服务器获取不到元数据**

  服务器只能接收到Cookie的名称和值，没有其他的信息。重要的是，它们不知道Cookie的来源。如果知道的话，服务器可以拒绝不是它们设置的Cookie。

- **对于安全Cookie缺少完整性验证**

  Cookie可以跨越HTTP和HTTPS工作是一个主要的问题。虽然你可以使用`secure`属性来指明一个Cookie只能通过加密通道提交，但是安全和不安全的Cookie都存储在同一个命名空间里。更糟糕的是，安全标志不属于Cookie身份识别的一部分；如果Cookie名称、域名和路径匹配，一个不安全的Cookie就可以覆盖掉之前的安全Cookie。

简单地说，HTTP Cookie的主要缺陷是它们的完整性没有得到保证。在本节的剩余部分，我将集中讨论TLS上Cookie设计的安全含义，为了更广泛地覆盖这个话题，包括覆盖不同应用的安全问题，我推荐你阅读Michal Zalewski的《Web之困：现代Web应用安全指南》（*The Tangled Web*，2011年由No Starch出版社出版）。

5.3.2 Cookie 篡改攻击

有三种Cookie篡改攻击：两种会导致新Cookie的创建，因此被归类于*Cookie注入*（cookie injection）；第三种使得Cookie被删除。按照应用程序安全的惯例，这些攻击都具有几分不寻常且戏剧化的名称。

在过去几年中，不同的研究者都发现了这些问题，并赋予它们不同的名称。虽然我更倾向于Cookie注入，因为这个名称准确地描述了到底发生了什么，但是你可能会遇到其他名称：*跨站Cookie*（cross-site cooking）①、*Cookie固定*（cookie fixation）、*Cookie强迫*（cookie forcing）②和*Cookie投掷*（cookie tossing）③。

---

① Cross-Site Cooking，http://www.securityfocus.com/archive/107/423375/30/0/threaded（Michal Zalewski，2006年1月29日）。

② Cookie forcing，http://scarybeastsecurity.blogspot.co.uk/2008/11/cookie-forcing.html（Chris Evans，2008年11月24日）。

③ New Ways I'm Going to Hack Your Web App，http://media.blackhat.com/bh-ad-11/Lundeen/bh-ad-11-Lundeen-New_Ways_Hack_WebApp-WP.pdf（Lundeen等，2011年8月）。

1. Cookie驱逐

Cookie驱逐（cookie eviction）是针对浏览器的Cookie存储的一种攻击。如果由于某种原因，攻击者不喜欢浏览器的Cookie存储中存放的Cookie的话，他可能会尝试利用Cookie存储对单独Cookie大小、每域名的Cookie数量和组合Cookie大小的限制来进行攻击。通过提交大量的假Cookie，攻击者最终可以让浏览器丢弃掉所有的真实Cookie，而将攻击者提交的假Cookie存储在浏览器中。

浏览器的Cookie存储空间在多个方面存在限制。Cookie的总数有限制，存储空间也是这样。还存在一个每域名限制（通常是几十个），这主要是防止单个域名占满整个存储空间。单独的Cookie一般被限制在4 KB左右。因此，一个Cookie驱逐攻击可能会使用多个域名来让Cookie存储空间完全溢出。

2. 直接Cookie注入

在执行直接Cookie注入的时候，攻击者面对的是一个使用安全Cookie的网站。因此，他无法直接看到Cookie（在不破解加密的情况下），但是可以创建新的Cookie或者覆盖已有的Cookie。这个攻击利用了安全和不安全Cookie存在于同一个命名空间[①]中的事实。

从概念上讲，这个攻击与5.2节所描述的Cookie窃取类似：攻击者截获受害者发出的任意明文HTTP事务并用它来强制向目标网站发起明文HTTP请求。然后攻击者截获请求并且在响应中包含任意的Cookie。攻击可能像下面这样简单：

```
Set-Cookie: JSESSIONID=06D10C8B946311BEE81037A5493574D2
```

在实际中，为了让覆盖生效，强制Cookie的名称、域名和路径必须与原始Cookie匹配。攻击者必须观察目标网站使用的元数据并在攻击中替换它们。例如，Tomcat设置的会话Cookie永远将路径设置成网站的根目录：

```
Set-Cookie: JSESSIONID=06D10C8B946311BEE81037A5493574D2; Path=/
```

3. 从相关主机名进行的Cookie注入

当直接注入Cookie不可能的时候（例如无法冒充目标网站），攻击者可以基于Cookie在相关主机名之间共享这一点展开攻击。如果攻击者能冒充其他相关主机名上的网站，他也许可以从那里进行Cookie注入[②]。

例如，www.example.com是一个强安全性的网站，但是此域名下还同时存在一个博客站点，位于blog.example.com并由第三方运营，安全性较差。如果攻击者能找到博客程序上的一个XSS漏洞，他就可以修改其Cookie。攻击方法与前一部分“直接Cookie注入”一样：受害者被迫向存在漏洞的网站提交一个HTTP请求，此网站会设置任意的Cookie。

---

① Multiple Browser Cookie Injection Vulnerabilities，http://www.westpoint.ltd.uk/advisories/wp-04-0001.txt（Paul Johnston和Richard Moore，2004年9月15日）。

② Hacking Github with Webkit，http://homakov.blogspot.ru/2013/03/hacking-github-with-webkit.html（Egor Homakov，2013年3月8日）。

> **注意**
> 当然，由不同实体或组织运营一个网站的时候，就格外值得引起注意。不仅其他组织的成员是潜在的薄弱环节，他们也可能是威胁本身。

如果受害者没有保存目标网站的任何Cookie，攻击者就很幸运了。无论他向受害者设置什么Cookie，受害者都会使用。假设存在XSS漏洞，攻击就可以像执行下面这条语句一样简单（从blog.example.com上的一个页面执行）：

```
document.cookie = 'JSESSIONID=FORCED_ID; domain=example.com';
```

注意攻击者是如何使用domain属性来将Cookie的作用范围从默认的blog.example.com扩展到example.com的，这将对预期目标www.example.com起作用。

5

- 取得第一个Cookie

通常，受害者已经持有了一些真实的Cookie。如果攻击者使用相同的名称注入另一个Cookie（像之前的例子中那样），浏览器会同时接受两个Cookie并将它们在同一个请求中一起发出：

```
Cookie: JSESSIONID=REAL_ID; JSESSIONID=FORCED_ID
```

这种情况之所以会发生是因为浏览器将这两个值当成不同的Cookie，它们的名称、域名和路径属性都不完全匹配。虽然攻击者成功地注入了一个Cookie，但是攻击没有效果。这是因为当多个Cookie的名称相同的时候，一般服务器端的应用程序只会使用第一个Cookie。

从这里开始，攻击者可以尝试来驱逐所有真实Cookie了，这是通过使用大量的假Cookie来实现的。这在理论上是可以实现的，但是要成功实施，还需要一些技巧。

作为一种选择，攻击者可以调整Cookie的元数据来将假的Cookie放在前面的位置。一种实现这种技巧的方法是使用path属性[①]，这利用了浏览器优先提交拥有更多属性的Cookie的特性：

```
document.cookie = 'JSESSIONID=SECOND_FORCED_ID; domain=example.com; path=/admin';
```

假设浏览器正在访问位于/admin/路径下的资源，它将按照如下顺序提交Cookie：

```
Cookie: JSESSIONID=SECOND_FORCED_ID; JSESSIONID=REAL_ID; JSESSIONID=FORCED_ID
```

如果目标有多个，攻击者可以提交多个Cookie，每个Cookie一个路径。但是也存在这样一种情况，当从一个相关网站处发起的Cookie可能会覆盖掉原有Cookie时，这会在目标网站显式地将Cookie的域设置成根主机名的时候发生（例如，example.com）。

- 使用相关主机名覆盖Cookie

从相关主机名覆盖Cookie并不总是奏效，因为大部分网站在设置Cookie的时候并不显式地指定域。这些Cookie被标识成host-only。当从一个相关域名注入Cookie的时候，你需要指定一个域，这意味着这个Cookie可能永远都不会匹配到原始的Cookie，即使主机名相同。

还有一个原因导致从相关主机名覆盖Cookie失败：你无法为兄弟主机名颁发Cookie。对于blog.example.com，你可以为example.com和www.blog.example.com颁发Cookie，但是不能为

① Understanding Cookie Security，http://kuza55.blogspot.co.uk/2008/02/understanding-cookie-security.html（Alex kuza55，2008年2月22日）。

www.example.com颁发Cookie。

这让我想到了两种可以成功覆盖的场景：

- 显式地将Cookie的域名升级成根域名的网站。我只使用Firefox 28进行了测试，但是其他绝大多数浏览器应该也都有相同的行为。
- 对于IE浏览器（基于版本11进行了测试），没有对显式和隐式设置的域名进行区分。然而，因为Cookie的名称总是要匹配的，这种攻击只对那些为根域名颁发Cookie的网站有效。

- 用假的相关主机名覆盖Cookie

还有一种场景攻击者可以对原始Cookie实现覆盖：网站显式地设置了Cookie的域，但该域名并非必须是根域。

这是因为中间人攻击者可以选择使用哪个相关主机名进行攻击。互联网的核心运行于未经身份验证的DNS上，这意味着攻击者可以劫持DNS并伪造主机名。例如，如果攻击者需要对www.example.com进行攻击，他可以伪造一个假的子域，例如，www.www.example.com。使用这个子域名，他可以为www.example.com颁发Cookie。

5.3.3 影响

说来有趣，很多网站都是基于攻击者不能发现或者影响Cookie内容这个前提而进行设计的。因为这个假设不成立，所以就会有隐患，但是具体如何进行攻击还依赖具体的应用。

- XSS

  如果开发人员认为Cookie不会变化，他们可能以不安全的方式使用。例如，他们可以将Cookie输出到HTML中，这种情况下信息的泄露可能会导致XSS漏洞的产生。
- CSRF防御绕过

  有些网站依赖跨站请求伪造（cross-site request forgery，CSRF）防御功能，在这种场景下，一个令牌字符串会被嵌入到页面中，其值与Cookie中的一致。如果可以更改浏览器中某个Cookie值的话，这个防御手段就没用了。
- 应用程序状态改变

  开发人员经常将Cookie当成一种可以抵御篡改的安全存储方法。有时候应用程序的某些环节会依赖Cookie的值来进行决策。如果Cookie可以进行修改，那相当于应用程序的行为也可以进行修改。例如，可能存在一个名为`admin`的Cookie，如果其值为`1`的话，表示当前用户为管理员。很明显，用户可以通过修改自己Cookie的值来实现攻击。因此严格地说，这并不是TLS相关的问题，然而这种问题依然会作为一种攻击向量而被中间人攻击者所利用。下面小节介绍的防御手段会针对所有这种类型的攻击进行防护。
- 会话固定

  会话固定（session fixation）是一种反向的会话劫持攻击。相对于获得受害者的会话ID，这种攻击是由攻击者将自己在目标网站上的会话ID强加给受害者使用。这种攻击与会话劫持相比，威力没有那么大，但是取决于目标网站提供的特性，也可能造成严重的后果。

5.3.4 缓解方法

Cookie篡改攻击总体来说可以通过采取合适的缓解步骤来处理，这些缓解的方法主要有防止攻击者伪造Cookie以及检查接收到的Cookie是否合法等。

- 使用HSTS并覆盖子域名

  HTTP严格传输安全（HTTP strict transport security，HSTS）[①]是一个相对较新的标准，用于强制使用加密来访问启用它的网站。可以将HSTS配置为对所有子域名启用。使用这个方法，中间人攻击在不攻破加密的情况下无法使用DNS欺骗来注入Cookie。

  HSTS极大地减小了攻击面，但是使用它并不是毫无问题。首先，并不是所有浏览器都支持HSTS。其次，它无法处理下面的情形：正确的（加密的）相关域名被盗用或者由不同的、不可信的实体运营。我将在10.1节中详细讨论HSTS。

- Cookie完整性验证

  抵御Cookie注入的最好方法是对Cookie进行完整性验证：确保从客户端收到的Cookie确实是由本网站设置的。这可以通过使用基于散列的消息验证代码（hash-based message authentication code，HMAC）实现[②]。

  不需要从JavaScript访问的Cookie可以通过额外的加密来进行保护。

  以这样的方式设计完整性验证方案是非常关键的：颁发给一个用户的Cookie对其他用户无效。否则，攻击者可以从网站上获取到一个合法的Cookie（用他自己的账号）并将其注入到受害者的账号中。

  Cookie完整性验证和加密并不能帮助保护会话Cookie，会话Cookie一般是使用一种基于时间限制的密码代替机制。通道ID是一种尝试解决这种问题的方法，该方法在浏览器和网站的TLS层面创建了一种密码学意义上的绑定。[③]这种方法也叫作*通道绑定*（channel binding），有效地创建了一个可以替换掉HTTP会话的会话。在实际中，现有的基于Cookie的机制会被保留，不过将与安全通道进行绑定以防御会话劫持。

5.4 SSL 剥离

SSL剥离（SSL stripping，更准确地说是HTTPS剥离）攻击利用了很多用户在开始访问某个网站的时候都是从明文部分开始或者在输入网站地址的时候没有显式地使用https://前缀（浏览器会优先使用明文连接）。因为明文流量是完全可见并存在风险的，所以这些流量可以被主动网络攻击者修改。

例如，如果网站的一个页面包含了一个安全连接，攻击者可以将此安全连接修改成明文连接。

---

① RFC 6797: HTTP Strict Transport Security，http://tools.ietf.org/html/rfc6797（Hodges等，2012年11月）。

② RFC 2014: HMAC: Keyed-Hashing for Message Authentication，http://tools.ietf.org/html/rfc2104（Krawczyk，1997年2月）。

③ TLS Channel IDs，http://tools.ietf.org/html/draft-balfanz-tls-channelid-01（Internet-Draft，D. Balfanz和R. Hamilton，2013年12月31日过期）。

由于没有了安全连接，受害者永远都无法使用加密访问网站。与此同时，攻击者可以在真正的网站和受害者之间进行代理，获取原本是加密的内容。这样一来，攻击者不仅可以看到敏感信息，还可以任意对请求和响应进行修改（如图5-3所示）。

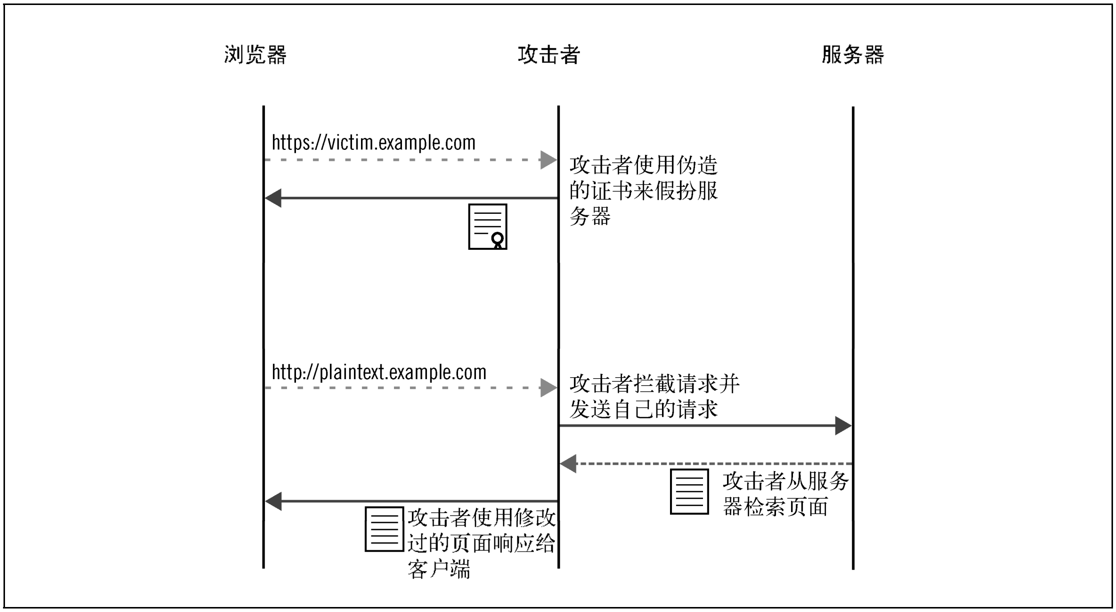


图5-3　中间人攻击的变种

HTTPS剥离攻击利用了大多数用户都无法区分安全和不安全浏览的事实。对于那些可以区分出区别的用户，攻击者可以使用一个狡猾的替代方案，那就是将用户重定向到一个被攻击者完全控制的安全网站，其名称和真实的网站十分类似。常见伎俩是使用包含整个目标地址的长地址（例如，https://victim.com.example.com）或者使用与真实网站只差一个字符或使用相似Unicode字符的地址。

在暗中，攻击者就可以与目标网站建立安全或者不安全的连接，但是这对于受害者来说已经没有意义，因为攻击者已经不仅可以看到原本加密的信息，还可以任意对其进行修改。

从攻击者的角度来看，HTTPS剥离攻击很有意思，因为这种攻击可以轻易地进行自动化（通过使用已经存在的现成工具）。此类工具中有一个很知名，叫作`sslstrip`[①]。

5.5　中间人攻击证书

HTTPS剥离攻击对于大部分用户可能是有效的攻击手段，但是也有一些情况会导致其不起作用。一些用户确实会注意到安全和不安全网站之间的差异，他们甚至会检查浏览器的锁符号或者

① sslstrip，http://www.thoughtcrime.org/software/sslstrip/（Moxie Marlinspike，2011年5月15日）。

（少见的）EV证书的绿条。一些用户还会将安全网站加入到浏览器书签中并从书签中直接访问。

位于通信两端中间的攻击者依然可以将全部流量进行重定向，但是成功的实施攻击也需要进行一番努力。可能的攻击手段有以下这些。

- 利用验证漏洞

  TLS的安全性依赖于客户端是否正确地验证了接收到的证书。如果验证逻辑的实现有问题，就可能会被攻击者使用无效的证书或者证书链。

- 伪造证书

  伪造证书（rogue certificate）是指假的CA证书，但是这些证书却被浏览器所接受。这种证书虽然很难获得，但是任然存在获取的可能性。例如，一个这样的证书在2008年针对RapidSSL展开的攻击中被伪造出来。你可以在4.5节中找到关于这次攻击的更多信息。另外的一种可能是强大的攻击者可以暴力破解较弱的1024位的CA证书的私钥。在2014年，仍然有很多安全强度较弱的证书受到主流浏览器的信任。据估计，破解一个1024位的私钥只需要花费一百万美元，但是这种破解需要将近一年的时间。[①]

  获取到伪造的CA证书之后，除了对安全最偏执的用户之外，其他用户根本无法察觉到攻击者。结合中间人攻击可以干扰OCSP作废检查以及大多数浏览器会忽略掉OCSP失败的情况，如果攻击者可以长时间控制受害者的互联网连接，也就没有办法对伪造的CA证书进行作废。

- 自签名证书

  如果之前的办法都不可行，攻击者也许可以尝试最简单的方法，这就是用一个复制真实证书各项值的自签名证书来欺骗受害者。这种证书会导致浏览器出现警告，但是多数用户都会忽略掉警告。更多的细节在下一节中说明。

此类型工具有两个很知名，分别是`sslsniff`[②]和`SSLsplit`[③]。

5.6 证书警告

为了获得真正的安全性，密码学要求身份验证。如果你无法确认你是在与正确的对方通信，那再讨论任何安全性都没有意义。某人可能正在劫持通信来伪造成你原本的通信对端，并且你还发现不了。这种情况就像是拿起电话之后，在没有确认对方是否是其声明的身份之前，就与对方开始讲话。

在TLS中，我们使用证书来进行身份验证（TLS支持其他的身份验证方法，但是很少使用）。在连接到一个服务器的时候，你会想到某个特定的域名，然后期望服务器会提供一个证书来证明

① Facebook's outmoded Web crypto opens door to NSA spying，http://www.cnet.com/news/facebooks-outmoded-web-crypto-opens-door-to-nsa-spying/（CNET，2013年6月28日）。

② sslsniff，http://www.thoughtcrime.org/software/sslsniff/（Moxie Marlinspike，2011年7月25日）。

③ SSLsplit - transparent and scalable SSL/TLS interception，https://www.roe.ch/SSLsplit（Daniel Roethlisberger，2015年3月16日）。

它有权利来处理这个域名的流量。

如果你收到了一个无效的证书，正确的做法是中断对网站的访问。不幸的是，浏览器并不这样做。因为网络上充满了无效的证书，几乎可以保证的是，没有任何一个无效证书是由于攻击导致的。面对这个问题，浏览器厂商在很久之前就决定不去强制严格SSL连接安全，而是将这个问题推给用户，也就是以证书警告（certificate warning）的形式；例子如图5-4所示。

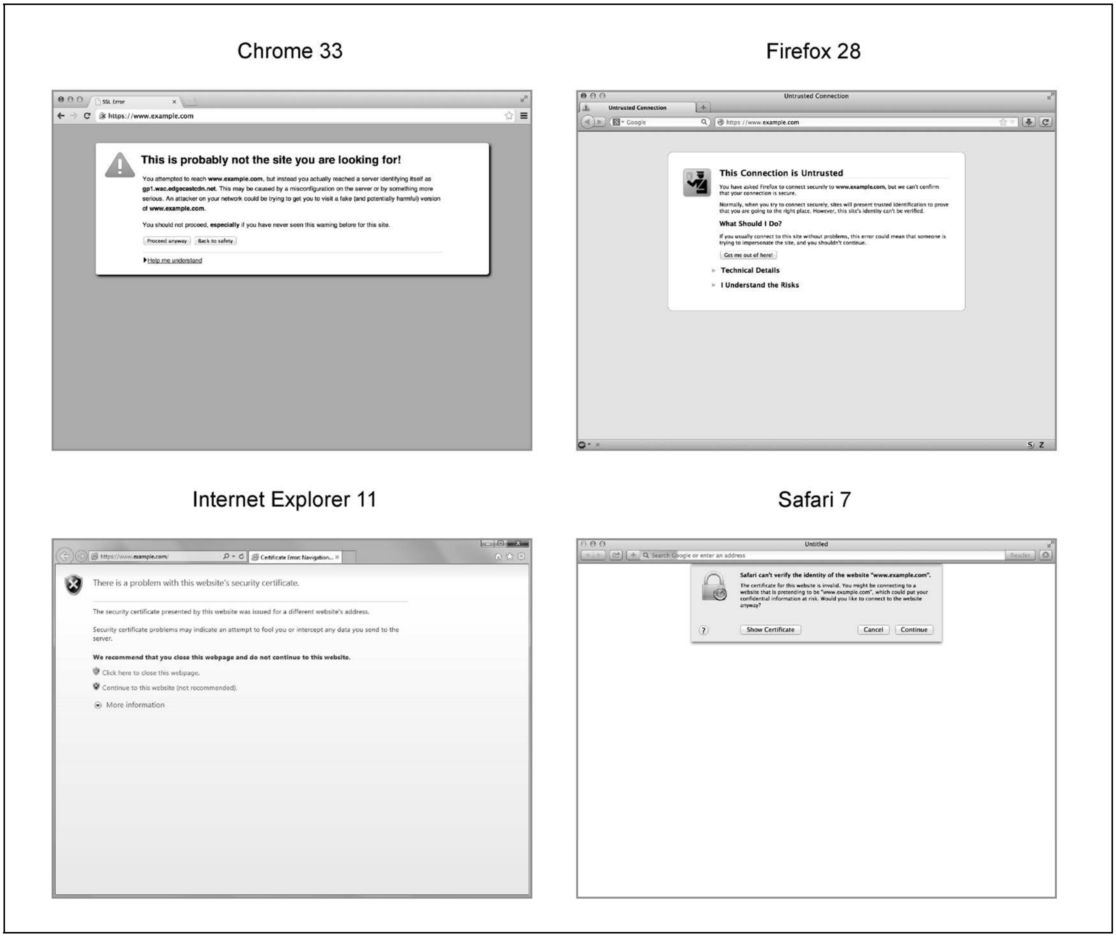


图5-4　浏览器中的证书警告例子

这让我了解到了TLS最丑陋的事实之一：TLS的核心目的是保护你不被中间攻击所危害，但是当真的攻击到来的时候，你所能得到的全部只是一个来自浏览器的证书警告，然后由“你”来决定是否遭受到了攻击。

5.6.1 为什么有这么多无效证书

有很多关于无效证书流行广泛的轶事证据，很难找到没有遇到过这些证书的人。以下是几个根本原因。

- 配置错误的虚拟主机

  今天，多数网站只在端口80上运行且不使用加密。一个常见的配置错误是让这些明文网站和在443端口上使用加密的其他网站使用同一个IP地址。这样一来，如果用户尝试使用https访问明文网站的话，就会导致错误的发生：证书和域名不匹配。

  这个问题的部分原因是，在技术层面，我们没有一种机制可以使得网站来声明是否支持加密。从这个角度来看，托管明文网站的正确做法是让它们使用关闭了端口443的IP地址。在2010年，我扫描了1.19亿个域名，寻找加密网站。[1]扫描的列表包括所有的.com、.net和.org域名。我发现2265万（19%）的安全网站托管在大概200万个IP地址上。在这些安全网站中，只有720 000个（3.2%）网站的证书和域名匹配。

  有一个名称正确的证书是一个好的开始，但是这还不够。大概30%的这类证书在2010年的调查中由于其他原因而实际是无效的（不可信）。

- 域名覆盖范围不足

  在少数场合下，网站管理员购买并部署了证书，但是证书中没有包含全部的网站域名。例如，你在www.example.com上运行了一个网站，证书可能包括这个域名以及example.com。如果网站还有其他的域名，证书中也需要包含那些域名。

- 自签名证书和私有CA

  自签名的证书以及私有CA签发的证书不适合在公开场合使用。这类证书无法简单并可靠地与中间人攻击的证书区分开。在我的调查中，大约48%的无效证书是因为这个原因。

  那么为什么人们要使用这类证书？原因有很多：(1) 购买、配置和续签证书带来了额外的工作，而且需要持续投入；(2) 直到几年前，证书的价格仍比较昂贵；(3) 一些人认为公共可信的证书应该是免费的，因而拒绝进行付费购买。然而，最简单的事实是公共可信证书对于公开网站是适合的。我们目前对此还没有替代品。

- 设备使用的证书

  当今，很多设备都有基于Web的管理界面，这些界面要求使用安全通信。当这些设备被制造出来的时候，域名和IP还无法确定，这意味着生产厂商无法安装有效的证书。理论上来讲，最终用户可以自己在设备上安装证书，但是很多这种设备都不常用，因此也就不值得去购买并安装证书。此外，很多设备的管理界面也不允许用户自己配置证书。

- 过期的证书

  另外一个无效证书产生的重要原因是过期。在我的调查中，57%的无效证书是由于过期导致的。在很多情况下，网站的管理员忘记去续签证书，或者放弃了取得有效证书的想法，

---

① Internet SSL Survey 2010 is here!，http://blog.ivanristic.com/2010/07/internet-ssl-survey-2010-is-here.html（Ivan Ristić，2010年7月29日）。

但是并没有将老的证书下线。

- 错误的配置

  另外一个常见的问题是错误的配置。为了让一张证书受到信任，每个浏览器都需要建立起一条信任链，从服务器证书到一个可信任的根证书。服务器需要提供除了根证书之外的整个信任链，但是根据SSL Pulse的数据，大约6%的服务器的证书链不完整。在某些情况下，浏览器可以对此作出变通，但是通常它们并不会这样做。

当谈到用户体验的时候，一个2013年的研究结果显示在39亿个公开的TLS连接中，有1.54%的连接存在证书警告。[①]但是这只是在公开的互联网上的情况，在这里网站一般都会尽量避免警告。在某些特定的场景下（例如内联网或者内部应用），你可以需要每天都需要点击通过证书警告，以便可以访问和工作相关的Web应用程序。

5.6.2　证书警告的效果

如果没有证书警告，世界将变得更加美好，但是实际上浏览器厂商一直在增强安全性和让用户满意两者之间寻找合适的平衡。在2008年，我曾试探性地去说服Mozilla从Firefox中取消为无效证书增加例外的功能，这将使得跳过证书警告变得特别困难。毫无意外，我的bug提交被拒绝了。[②]他们的答复（以指向早年间一个帖子的链接的形式[③]）是他们尝试过这样做，但是用户对此的反抗十分强烈。这反映了一个更加广泛存在的目标不一致问题：浏览器厂商希望增加市场份额，但是增强安全会阻碍这一目标。这样一来，浏览器厂商只能在确保大多数用户高兴的前提下，尽可能多地提供安全性。在某些非常偶然和少见的时候，用户才会对来自真正的中间人攻击导致的证书警告进行抱怨，这提醒了每一个人证书警告的真正含义。[④]也许关于中间人攻击的最大问题是用户无法感知它们（毕竟证书警告是生活中的“正常”部分），也不会报告这些攻击。

然而，事实是让用户越难忽略掉证书警告，为用户提共的安全性就越强。今天，主流的浏览器都使用了所谓的*间隙性警告*（interstitial warning）或*中断性警告*（interruptive warning），这些警告会覆盖整个浏览器的内容窗口。老式的弹对话框的警告方式（Safari还在使用）被认为是无效的，它们看起来与我们从系统中经常看到的其他对话框没什么区别。大部分浏览器允许用户点击忽略掉警告。当可以简单地一次点击就绕过障碍时，在用户和网站之间唯一剩下的就是那些严厉警告的语言了。像我们预料的那样，很多人都会选择继续浏览。

早期关于证书警告有效性的报告显示出了非常高的点击率，但是这个大部分是在可控的实验

---

① Here’s My Cert, So Trust Me, Maybe? Understanding TLS Errors on the Web，https://www.icsi.berkeley.edu/pubs/networking/ICSI_heresmycert13.pdf（Akhawe等，WWW Conference，2013年）。

② Bug 431827: Exceptions for invalid SSL certificates are too easy to add，https://bugzilla.mozilla.org/show_bug.cgi?id=431827（Bugzilla@Mozilla，报告于2008年5月2日）。

③ TODO: Break Internet，http://blog.johnath.com/2007/10/11/todo-break-internet/（Johnathan Nightingale，2007年10月11日）。

④ Bug 460374: All certificates show not trusted - get error code (MITM in-the-wild)，https://bugzilla.mozilla.org/show_bug.cgi?id=460374（Bugzilla@Mozilla，报告于2008年10月16日）。

室环境中得出的，这引入了一些不可靠的因素。[①]

> 此外，我们的分析也引起了对于人类行为安全研究可用调查结果的实验室研究仍寥寥无几现状的担忧，尤其是在生态验证很重要的情况下……参与我们试验的那些关心安全的人所表现出来的不愿意的情绪，引起了人们对这类针对安全实践和主要人口行为的研究结论的准确性和可靠性的担忧。

与此同时，浏览器厂商开始使用遥测（telemetry）技术来监控产品的使用情况。这允许对用户在自己环境中的使用情况进行观察，提供了更加准确的结果。实验表明，Firefox提供了最好的实现方法，只有33%的用户继续访问了无效证书的网站。作为对比，大概70%的Chrome用户会选择继续浏览。[②]一个稍后的调查结果中反映出Chrome在模仿了Firefox的实现之后，点击率下降到了56%。[③]

5.6.3 点击-通过式警告与例外

Firefox对无效证书处理方法的成功也可以从这个角度来解释，那就是Firefox是唯一不使用点击-通过式警告的浏览器。相反，它会让你经历一个多步骤的过程来创建一个证书例外（certificate exception），创建之后无效的证书会被当成可信证书来处理，这对后续的访问也是同样有效的。在这种情况下我们可以预料到，这个过程的每一个步骤都会让一些用户放弃继续并留意到警告。

反对证书例外的论点是它会使自签名证书的使用变得更加容易。确实如此，但这也不一定就是坏事。在使用者事先了解的情况下，自签名证书本身不一定就不安全。例如，我家里有一个ADSL路由器，我可以通过TLS访问它的管理界面。我清楚地知道它没有有效的证书，但是我没必要每次都在登录管理界面的时候点击并通过一个证书警告。此外，例外是基于每证书而创建的。这也就是说，如果我受到了攻击，证书警告会再次出现。在安全领域，这个方法叫作首次使用信任（trust on first use，TOFU），该方法已在全世界上千万服务器的SSH服务上成功使用。这种方法的另外一个名称是密钥连续性管理（key continuity management）。

证书例外仅对于个人使用以及知道只有在安全的情况下才可以创建例外的那一小部分技术用户来说是有用的。很关键的一点是，证书例外必须要在用户没有遭受攻击的情况下创建。在我的例子中，我知道ADSL路由器上的证书自己不会改变，如果在创建例外之后再次出现了警告，那将是非常不正常的。

---

① On the Challenges in Usable Security Lab Studies: Lessons Learned from Replicating a Study on SSL Warnings，https://cups.cs.cmu.edu/soups/2011/proceedings/a3_Sotirakopoulos.pdf（Sotirakopoulos等，Symposium on Usable Privacy and Security，2011年）。

② Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness，https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/akhawe（Akhawe和Felt，USENIX Security，2013年）。

③ Experimenting At Scale With Google Chrome's SSL Warning，http://www.robreeder.com/pubs/sslExperimentCHI2014.pdf（Felt等，ACM CHI Conference on Human Factors in Computing Systems，2014年）。

5.6.4　缓解方法

如果你在乎你网站的安全，那就一定非常担心你的用户会在真正的中间人攻击的时候执行点击–通过操作。毕竟，你熬过了所有的麻烦，包括使用有效证书、配置服务器以及其他所有在你这里能做到的保护用户安全的事情。

显然，对于整个生态系统，你能做的事情很少，但是你可以通过使用HSTS来保护你的网站，这将是一个对于浏览器发出的信号，来让它们调整行为并使用更加严格的安全姿态来面对加密。HSTS的一个特性是不允许证书警告。如果启用了HSTS的网站出现了证书问题，所有的错误都会被当成严重级别并且无法忽略。在这种情况下，你就可以重获对安全的控制。

5.7　安全指示标志

安全指示标志是一种浏览器的UI元素，用来表示当前页面的额外安全信息，一般来说会表示下面4种含义中的一种。

- “当前页面使用SSL”
- “我们知道运营此网站的合法实体的身份”
- “这个页面使用的是无效证书”
- “页面中的部分内容不是加密的”

除了EV证书（EV证书将网站与某个合法实体进行关联），其他的指示标志之所以存在，主要是因为网站加密是可选的，并且浏览器对于安全处理比较随意（如图5-5所示）。如果Web是100%加密的，并且不存在证书警告以及混合内容的情况，你可能只需要关心是否有EV证书的标志就行了。

安全指示标志最大的问题是大多数用户都不关注它们，甚至根本没有注意到它们的存在。我们是通过一些针对安全指示标志的研究而了解到这个情况的，其中一个研究使用了眼球追踪技术。该研究发现大多数用户很少看安全标志，主要的目光都集中在内容上。[①]在此研究中，没有任何参与者注意到了EV标志。这证实了另外一个研究的结论，该研究的作者得出了完全相同的结论。[②]

也许造成这种结论的一个原因是安全指示标志缺少一致性，无论是在不同的浏览器之间还是在同一浏览器的不同版本之间。浏览器的指南对此有说明，但是不够详细。[③]

① Exploring User Reactions to New Browser Cues for Extended Validation Certificates，https://www.ccsl.carleton.ca/paper-archive/sobey-esorics-08.pdf（Sobey等，ESORICS，2008年）。

② An Evaluation of Extended Validation and Picture-in-Picture Phishing Attacks，http://www.adambarth.com/papers/2007/jackson-simon-tan-barth.pdf（Jackson等，Proceedings of Usable Security，2007年）。

③ Web Security Context: User Interface Guidelines，http://www.w3.org/TR/wsc-ui/（W3C建议，2010年8月12日）。

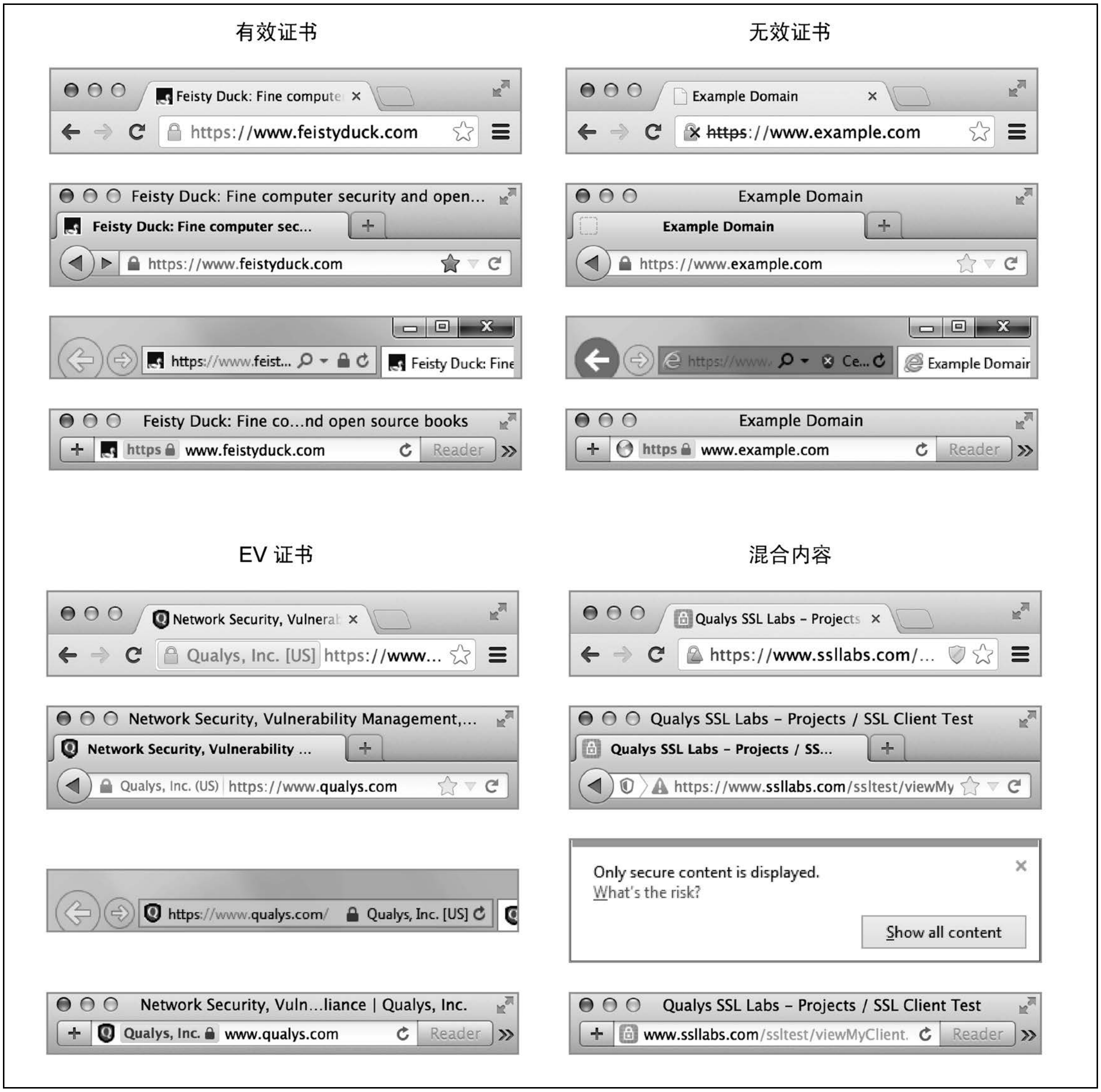


图5-5 当前浏览器安全指示标志的例子

我想起了在SSL刚被使用的时候，为了让用户了解“锁”符号的含义（“如果你看见一个锁，你就是安全的”），进行了大量的宣传教育。几年之后，浏览器厂商开始随意摆弄用户界面。在某些情况下（例如Firefox），可能每个版本之间都存在差异。

与此同时，网站开始在网页上使用锁标记，这进一步削弱了消息的传达。因此锁标记的含义从具体的（使用加密）变成了普遍的安全指示标志。在很多情况下，锁标记的出现是没意义的。例如，很多网站明显地提供了锁标志，却不使用加密。

今天，仅从广义上来讲，仅有的一致性是EV证书的绿色使用。这一点目前是所有浏览器都遵守的。

对于移动平台，情况看起来更糟。由于屏幕尺寸更小，浏览器厂商努力地去移除几乎所有的页面元素，而这也会影响到安全指示标志。对于很多移动浏览器，即使是安全专家也需要很费劲才能区分开安全和不安全网站。[①]

这导致一些研究者得出了移动用户遭受钓鱼攻击威胁的可能性是其他用户的三倍。[②]此外，移动应用（非浏览器）的安全性总体来说也很难评估。虽然所有应用都应该使用安全连接与服务器进行通信，但是实际情况是否这样我们无法得知，因为它们不提供安全指示标志。而且，即使它们这样做了，谁又能知道它们是否只是显示了一个锁标记但是实际上却依然没有加密通信呢？

5.8　混合内容

TLS协议将自己考虑在一个连接之内并且只关注数据在网络层的安全。这种分开的考虑对于简单的上层协议来说是奏效的，例如SMTP。然而，某些协议在同一个安全上下文（例如浏览器会话）中具有多个连接（例如FTP和HTTP）。TLS对这种情况没有提供任何的指导，这依赖于用户代理的开发人员进行安全方面的实现。

对于HTTPS，你可能很难找到一个只使用单一连接的页面。对于几乎所有的网站，HTML标记、图片、样式表、JavaScript以及其他页面资源不仅是从多个连接获取到，也有可能是来自互联网上不同的服务器和其他网站。为了将一个页面进行正确的加密，所有的页面资源都通过HTTPS进行获取是必要的。在实际中，全部使用HTTPS的情况很不常见，这就导致了混合内容（mixed content）安全问题的发生。

> **注意**
> 本节只覆盖了同页面混合内容的问题，但是在整个网站层面也存在同样的问题。网站如果混合这些安全和不安全的页面，就容易遇到开发问题（例如，使用不安全的Cookie或者不加密交付敏感内容）以及容易遭受SSL剥离攻击。

5.8.1　根本原因

为了理解混合内容为什么如此普遍，我们要首先回顾一下Web的根源并考虑它极快的进化速度。人们注意的焦点是如何将事情完成，因此要克服由成本、技术和安全带来的限制。

- 性能

  在SSL发展的早期，与明文HTTP相比其性能极差。今天，服务器有了更快的处理器和更

① Measuring SSL Indicators on Mobile Browsers: Extended Life, or End of the Road，http://www.cc.gatech.edu/~traynor/papers/amrutkar-isc12.pdf（Amrutkar等，Information Security Conference，2012年）。

② Mobile Users Three Times More Vulnerable to Phishing Attacks，https://securityintelligence.com/mobile-users-3-times-more-vulnerable-to-phishing-attacks/（Mickey Boodaei，2011年1月4日）。

多的内存，我们仍然在关注密码学操作的速度。回到早年的时候，取得高性能SSL的唯一方法就是使用特定的加速硬件，这些硬件都十分昂贵。

因为性能的问题，人们都选择远离SSL。对于将整个网站覆盖100%的加密是完全不可能的。你可能会争论说这种方法是有道理的，并且最终的选择会在部分安全和没有安全之间。

今天，性能依然是一个需要关注的问题，但是更多的是关于延迟。由于建立安全连接引入的多余RTT，访问安全网站会存在微小的延迟。

- Mashup

  从某种角度来说，Web确实高速发展起来了，Mashup（混合Web应用程序）的概念也随之诞生。网站不再是自己提供全部内容。相反，它们混合其他网站的内容并将精力用在用户体验上，而内容来源则被隐去。在某些情况下，内容是免费获得的，在其他情况下，Mashup操作是通过商业交易完成的。

  一种Mashup的场景是使用第三方的代码来做网站分析，尤其是Google在免费推出它的分析服务之后，这种现象变得十分普遍。根据某些评估所称，有50%的网站中都使用了Google Analytics。[①]

  Mashup于对安全来说简直就是噩梦。它们主要通过引入第三方的JavaScript代码来实现。不幸的是，虽然这个方法导致网站建设的成本急剧减小，但是也给了提供代码的第三方网站全面控制其他网站的可能。它同样给网站用户带来了一些问题：在同一个网站上集合众多实体，他们很难了解到底是在与谁通信以及数据存储到了哪里。

  在加密的上下文中，主要的问题是在很多场景中第三方的内容和服务不支持加密。有的时候，可以使用加密的服务但是价格更贵。这样一来，人们简单地在“安全”网站中去包含不安全（明文）内容。

  为了阐明这个问题，让我们看一下Google的广告平台AdSense，它从2013年9月开始才提供HTTPS服务。[②]

- 基础设施成本

  随着网站间竞争的加剧，如果网站只在一个地理位置上提供服务是无法有很强竞争力的。内容分发网络（content delivery network，CDN）为此崛起，它将内容以最优的方式交付给用户。主要原理是在全球部署大量的服务器，网站的访问者可以访问最快的那个。

  CDN的问题是其为很多用户提供了大量数据文件（通常是静态的）。加密不仅增加了CPU和内存的需求，同时也影响了缓存并增加了证书和私钥管理的负担。

  最重要的是IP地址的问题。对于明文的HTTP来说，因为虚拟主机的使用，IP地址不是问题。这使得大范围的托管和分发变得简单。安全网站的虚拟主机完全不同，这对于公开网站还是不可行。这意味着你需要去为网站分配独立的IP地址。你不得不将你的基础设施

① Usage statistics and market share of Google Analytics for websites，http://w3techs.com/technologies/details/ta-googleanalytics/all/all（W3Techs，2014年7月15日）。

② Use AdSense on your HTTPS sites，http://adsense.blogspot.co.uk/2013/09/use-adsense-on-your-https-sites13.html（Sandor Sas，2013年9月16日）。

拆分成很多组，这导致了更复杂的结构和运维成本的增加。

此外，IPv4地址的全球性短缺也是一个问题。一些公司通过使用共享的多域名证书来为不相关的网站提供服务，但这仍然很复杂。

基本的观点是，安全的CDN是有可能的，但成本较高。

基于上述原因，浏览器一般对页面加密只提供很少的完整性检查。在我们的开发文化中，混合内容问题得到允许并变得根深蒂固。

5.8.2　影响

混合内容问题造成的影响主要与内容没有被加密有关。经过很多年之后，形成了两个概念：*混合被动内容*（mixed passive content）和*混合主动内容*（mixed active content）。前者也叫作*混合显示*（mixed display），它们是低风险的内容，例如图片。后者也叫作*混合脚本*（mixed scripting），它们是高风险的内容，例如HTML和JavaScript脚本。

混合主动内容是真正的威胁来源。一个对于JavaScript文件不受加密的引用可能会被主动攻击者劫持，并用来获取对页面的完全控制，以及使用受害者的身份在网站上执行任意的动作。对于其他危险的资源类型也是这样，例如HTML标记（包括通过框架）、样式表、Flash和Java应用程序等。

虽然混合被动内容不像混合主动内容那样危险，但是仍然会破坏页面的完整性。在最不危险的情况下，攻击者可以通过在图片中插入信息来玩弄受害者。这可能会导致钓鱼攻击的发生。针对浏览器的图像处理代码，同样有可能对图片注入恶意代码。最终，某些浏览器可能会使用*内容嗅探*（content sniffing）并且可能会将图片当作脚本来执行，在这种情况下，攻击者也可能会控制整个页面。

此外，任何与主页在一个域名下的其他未加密内容都会泄露网站的会话Cookie。就像我在本章前面部分所述，没有正确保护起来的Cookie会被任何主动攻击者获取到，但是针对混合内容的场景，Cookie也会被被动攻击者获取到。

5.8.3　浏览器处理

在开始的时候，混合内容为所有的浏览器所允许。浏览器厂商希望网站的设计者和程序员可以理解这个潜在的安全问题并作出正确的处理。随着时间的推移，浏览器厂商改变了态度，开始对这个问题更加感兴趣并且对其进行了一些限制。

今天，大多数浏览器趋向于在安全和破坏网页浏览两者之间作出妥协：混合被动内容是允许的，混合主动内容则不允许。唯一美中不足的是，对于什么是主动内容的认定，各个浏览器都不相同。

- Android浏览器

  混合内容没有任何限制。

- Chrome

Chrome在版本14中修改了对于混合主动内容的处理，[①]但是直到版本21的时候才调整完毕[②]。从版本38起，Chrome将会阻止所有的混合主动内容。[③]

- Firefox

  Firefox很久之前就可以对混合内容进行检测并发出警告，但是由于内部实现的问题，没有办法来进行阻断。这个bug存在了大约12年之久。[④]在版本23中，Firefox终于开始对所有的混合主动内容进行阻断。[⑤]

- IE

  IE浏览器从至少IE5（1999年）开始就支持混合内容的检测。当IE浏览器检测到在同一个页面上存在加密和明文内容的时候，它将提示用户来决定如何处理。Microsoft有一次几乎就要实施对不安全内容进行默认阻断（带有提示消息），而且在IE7 beta版中已经这样做了，[⑥]但是由于用户的压力最终没有采用此功能。他们后来在IE9的时候才引入了这个功能。[⑦]与此同时，他们也开始对混合被动内容采取了默认允许的行为。

- Safari

  Safari目前不阻断任何混合内容，这使得它与其他主流浏览器相比显得区别很大。实际上，对于混合内容的问题，Safari还存在倒退的情况。在OS X的Safari 6中，曾经存在一个让用户来选择进行混合内容阻断的选项，但是这个选项在Safari 7中去掉了。

表5-1详细描述了当今主流浏览器对混合内容的处理方式。

表5-1　主流浏览器的混合内容处理方式，“是”表示允许混合内容（2015年3月）

	Images	CSS	Scripts	XHR	WebSockets	Frames
Android Browser 4.4.x	是	是	是	是	是	是
Chrome 41	是	否	否	否	否	否
Firefox 30	是	否	否	否	否	否

① Trying to end mixed scripting vulnerabilities，http://googleonlinesecurity.blogspot.co.uk/2011/06/trying-to-end-mixed-scripting.html（Google Online Security blog，2011年6月16日）。

② Ending mixed scripting vulnerabilities，http://blog.chromium.org/2012/08/ending-mixed-scripting-vulnerabilities.html（Google Online Security blog，2012年8月3日）。

③ PSA: Tightening Blink’s mixed content behavior，https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/Uxzvrqb6IeU/9FAie9Py4cIJ（Mike West，2014年6月30日）。

④ Bug 62178: Implement mechanism to prevent sending insecure requests from a secure context，https://bugzilla.mozilla.org/show_bug.cgi?id=62178（Bugzilla@Mozilla，报告于2000年12月6日）。

⑤ Mixed Content Blocking Enabled in Firefox 23!，https://blog.mozilla.org/tanvi/2013/04/10/mixed-content-blocking-enabled-in-firefox-23/（Tanvi Vyas，2013年4月10日）。

⑥ SSL, TLS and a Little ActiveX: How IE7 Strikes a Balance Between Security and Compatibility，http://blogs.msdn.com/b/ie/archive/2006/10/18/ssl-tls-amp-a-little-activex-how-ie7-strikes-a-balance-between-security-and-compatibility.aspx（Rob Franco，2006年10月18日）。

⑦ Internet Explorer 9 Security Part 4: Protecting Consumers from Malicious Mixed Content，http://blogs.msdn.com/b/ie/archive/2011/06/23/internet-explorer-9-security-part-4-protecting-consumers-from-malicious-mixed-content.aspx（Eric Lawrence，2011年6月23日）。

（续）

	Images	CSS	Scripts	XHR	WebSockets	Frames
Internet Explorer 11	是	否	否	否	否	否
Safari 8	是	是	是	是	是	是

如果你对你喜爱的浏览器的行为感到好奇，SSL Labs提供了对于浏览器的混合内容问题的测试[①]，可以进行参考。

> **注意**
> 混合内容的漏洞可能隐藏得非常深。在很多现代浏览器中，存在着很多方法使得不安全的HTTP请求从安全页面上发出。例如，浏览器插件可以发出任何请求而忽略掉宿主页面的加密状态。对于像Flash和Java这类插件尤其如此，因为它们都是有自己权限的平台。现在W3C在努力来推动浏览器对于混合内容处理的标准化，这应当可以使得所有产品的行为变得一致。[②]

5.8.4　混合内容的流行程度

说来有趣，混合内容非常常见。2011年，我们在Qualys调查会导致Web应用程序加密被完全破坏的应用层面的其他几个问题时[③]，一起调查了混合内容问题。我们分析了来自Alexa的前100万网站名单中的约250 000个网站的首页，发现其中22.41%的网站含有不安全内容。如果不算图片的话，那么比例将降低到18.71%。

2013年，我们针对Alexa前100 000名网站中的18 526个网站展开了一次更详细的研究。[④]对于每个网站分析了多达200个安全页面，共计481 656个页面。你可以在表5-2中看到研究的结论。

表5-2　Alexa排名前100 000名的网站中的481 656个安全页面的混合内容问题（来源：Chen等，2013年）

	引用数量	远程引用比例	文件数量	页面数量	网站比例
图像	406 932	38%	138 959	45 417	30%
Frame	25 362	90%	15 227	15 419	14%
CSS	35 957	44%	6680	15 911	12%
JavaScript	150 179	72%	29 952	45 059	26%
Flash	1721	62%	638	1474	2%
总计	620 151	47%	191 456	74 946	43%

① SSL/TLS Capabilities of Your Browser，https://www.ssllabs.com/ssltest/viewMyClient.html（SSL Labs，检索于2015年3月22日）。

② W3C: Mixed Content，https://w3c.github.io/webappsec/specs/mixedcontent/（Mike West，检索于2015年6月14日）。

③ A study of what really breaks SSL，http://blog.ivanristic.com/2011/05/a-study-of-what-really-breaks-ssl.html（Michael Small和Ivan Ristić，2011年5月）。

④ A Dangerous Mix: Large-scale analysis of mixed-content websites，http://www.securitee.org/files/mixedinc_isc2013.pdf（Chen等，Information Security Conference，2013年）。

> **注意**
> 即使所有的第三方链接都是加密的，也仍然存在使用其他网站的主动内容可能会被这些网站获取控制的问题。当下太多的网站包含任意的控件，而却从来不会从安全影响的角度进行思考。[①]

5.8.5 缓解方法

好消息是，尽管浏览器对于混合内容问题的态度是松懈的，但是你可以自己完全解决这个问题。如果你将你的网站实现得正确，就不会受到威胁。当然，说起来容易做起来难，尤其是对于大型的网站。

5

有两种技术可以使你最小化甚至是彻底消除混合内容问题，即使对于不正确实现的网站来说也是如此。

- HTTP严格传输安全

  HSTS是一种强制浏览器获取安全资源的机制，即使在面对用户错误（例如尝试通过80端口访问你的网站）以及实现错误（例如你的网站开发人员在安全页面上放了一个不安全链接）也依然有效。这个特性有效消除了混合内容的问题，但是它只能在你能控制的域名下工作。[②]最近有个说法称要改变这个特性：将来只要在一个页面上运用HSTS，所有的混合内容都会被消除，哪怕它存在于第三方的托管域名中。

- 内容安全策略

  为了阻断从第三方网站获取到的不安全资源，可以使用*内容安全策略*（content security policy，CSP）。这个安全特性可以对不安全资源进行阻断。它同时还有很多其他有用的特性来处理应用层安全问题。

HSTS和CSP都是声明式的方法，也就是说它们可以在Web服务器层面进行添加而不需要对应用进行修改。从某种角度来说，你可以将它们看作安全网，因为它们可以对没有正确实现的网站强制采取安全措施。

例如，安全网站上一个常见的问题来自于这样一个事实：很多网站都实现了80端口到443端口的自动跳转。这是合理的，因为如果某些用户确实访问了你的网站的明文部分，你会希望将他们送到正确（安全）的地方。然而，因为重定向是自动的，因此不可见，例如一个图片的明文链接被重定向到安全的链接，浏览器会在用户感知不到的情况下自动获取这个图片，然而攻击者可能会了解到这个行为。对于这个原因，可以考虑一直重定向到安全网站的同一个入口，如果你这么做了，在引用资源上的任何错误，将会被发现并且在开发阶段就得到解决。

当然，使用了HSTS的网站无法被利用进行攻击，因为浏览器自动对不安全链接进行了安全

---

① You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions，https://lirias.kuleuven.be/bitstream/123456789/354587/1/fp028-nikiforakis.pdf（Nikiforakis等，Computer and Communications Security，2012年）。

② Block Mixed Content for HSTS domains，https://groups.google.com/forum/#!msg/mozilla.dev.security/rR3SBOFvUqM/GtFai0Zt1TcJ（mozilla.dev.security讨论主题，2015年2月17日）。

替换。但是，从另外一个角度看，当前还不能完全寄希望于所有浏览器都支持HSTS，所以最好的办法就是减少这类混合内容的错误。

5.9　扩展验证证书

扩展验证（extended validation，EV）证书是证书的一个特殊分类，它将一个域名与一个合法实体进行了关联（个人无法申请EV证书）。在SSL的早期，所有证书都是经过严格验证再签发的，这与今天的EV证书很类似。证书的价格战导致了域名验证（domain-validated，DV）证书的广泛使用，DV证书主要依赖廉价的电子邮件验证。这种情况是可能发生的，这是因为业界并没有针对证书验证流程的正式规则。EV证书在2007年由CAB论坛所定义。[1]

EV证书有两个主要优势：(1) 域名持有者的身份是已知的，并且被写入证书中；(2) 人工验证方式使得证书的伪造变得困难。据我所知，还从来没有一个假的EV证书出现。

不过，上述这些优势在实际中是否有意义，则是一个问号，至少在考虑到整体的用户数量方面来看是这样的。就像本章前面的小节所讲，用户很少注意到安全指示标志，即使EV证书的标志如此显著的情况下也是如此。因此，最终用户将错过对域名持有者合法性的了解。此外，伪造的DV证书也可以用来对EV网站进行攻击。解决问题的唯一方法是让最终用户理解EV证书的含义，并记住使用EV证书的网站，然后注意到安全指示标志的缺失并最终决定不再继续访问。这看起来很不现实，因为很多用户即使在看到明显的证书警告的情况下也会继续访问网站。

对待EV证书的方法在未来仍可能会进行改进。例如，客户端可能会让网站只获取EV证书，就像现在你总是可以使用HSTS进行加密一样。

另外的一个问题是，EV证书只在页面层面被检测到并显示出来，而对于资源所使用的证书类型则没有进行处理（例如脚本）。基于EV证书的高成本，网站经常在大量不可见的二级域名上使用DV证书的情况，也是很常见的。[2]

这意味着一个细心的网络攻击者可以使用DV证书来攻击EV网站，而不会破坏掉绿色的安全指示标志。Zusman和Sotirov演示了一些有趣的攻击向量。[3]

- 其他域名的资源

  在很多情况下，网站会对主域名使用EV证书，但从其他很多使用DV证书的域名上获取资源。浏览器和这些使用DV证书的连接可能会被攻击者使用伪造的DV证书拦截，从而进行恶意注入。

- Cookie窃取

  因为浏览器不强制证书的连续性，使用一个DV证书来拦截主域名的连接是可能的，窃取

① EV SSL Certificate Guidelines，https://cabforum.org/extended-validation/（CAB论坛，检索于2014年7月15日）。

② Beware of Finer-Grained Origins，http://seclab.stanford.edu/websec/origins/fgo.pdf（Jackson和Barth，Web 2.0 Security and Privacy，2008年）。

③ Sub-Prime PKI: Attacking Extended Validation SSL，http://www.blackhat.com/presentations/bh-usa-09/SOTIROV/BHUSA09-Sotirov-AttackExtSSL-PAPER.pdf（Zusman和Sotirov，Black Hat USA，2009年）。

现有Cookie或者设置新的Cookie，之后再重定向回真实服务器。这种攻击进行得很快，因此多数用户都无法察觉。

- 持续恶意注入

  如果使用了缓存（本质上攻击者可以通告资源不过期），注入的恶意代码可以持续地保存在浏览器的缓存中，并在相当长的时间内起作用，即使在后续对网站的访问上也是如此。

5.10 证书吊销

对于证书的有效期，在两方面存在着矛盾，那就是人们既希望减少管理的成本又需要证书可以提供更多的更新信息。理论上来讲，在被认为可信之前，每张证书都需要检查吊销信息。但是在实际中，存在着大量的问题使得证书吊销变得很困难。

5.10.1 客户端支持不足

可以这样说，对于证书吊销的最大问题在于客户端的支持不足。你要么根本不需要证书吊销，要么严重依赖它，这使得情况变得更加糟糕。因为这样一来，总是可以“推迟”对它的处理。

我们很难了解到浏览器在处理吊销的时候做了哪些事情，包括什么时候检查吊销以及如何检查。因为没有文档，只能依赖挖掘邮件列表、bug报告以及阅读源代码来理解究竟发生了什么。例如，有轶事证据表明浏览器不会对中间证书进行吊销检查。在很长的时间里，我们并不清楚很多浏览器并不使用CRL（证书吊销列表）。对于新特性（例如OCSP stapling）的支持也是缓慢的。总结起来，这个领域就是一个大黑盒。通过测试也许可以得到一些答案，但是这只在某个时间点有效，对于某个产品的下个版本是否还是会有相同的行为则无法保证。

除了浏览器之外，命令行工具仍然深陷证书验证的泥潭，更无暇估计吊销的问题。并且因为大多数的SSL库默认不使用吊销检查，开发人员一般也就不会关心它了。

最后总结起来，就是由于这样或那样的原因，证书吊销没有起到应起的作用。

这种情况在2011年变得异常明显，当时有几个CA遭到入侵。在每个案例中，唯一有效的吊销欺诈证书的方法就是使用黑名单，但不是通过CRL或者OCSP。相反，所有的浏览器厂商通过发布补丁版本来解决问题，这些版本中包含对于欺诈证书的硬编码信息。Chrome、Firefox和Microsoft使用了特殊的机制来向浏览器推送证书黑名单，而不需要用户进行软件升级。

5.10.2 吊销检查标准的主要问题

从高层次来看，CRL和OCSP两者均存在设计上的缺陷，这使得它们的用处变得不大。这里主要有以下三个问题。

- 证书与查询之间不关联

  CRL和OCSP基于序列号来查找证书，这些序列号就是CA随意分配的一些数字。这很不幸，因为你无法确保你查询的证书就是CA所查询的证书。这种情况可以在CA遭受攻击之后被利用来创建伪造的证书，并让此伪造的证书包含一个现有的有效证书的序列号。

- **黑名单而不是白名单**

  从定义上来讲，CRL是一个黑名单，不能是其他东西。诞生于CRL之后的OCSP，在CRL框架之上被设计出来，目的是更加容易使用。早些年，OCSP响应程序通过添加CRL中存在的信息来进行工作。这错失了一次从黑名单切换到白名单的机会，包括检查证书是否有效，而不仅仅是检查是否吊销。

  对于黑名单的关注最终引起反响的是将“正常”的OCSP响应状态视为“未吊销”这一实践，即使在服务器并不知道序列号问题的情况下。在2013年8月，CAB论坛禁止了这一实践。

  这听起来有一些区别，但是这个设计上的缺陷的确在DigiNotar事件期间造成了一个真正的问题。因为这个CA被完全攻破了，没有任何关于哪些欺诈证书被签发了的记录。这样一来，这个CA的证书无法被单独吊销。虽然最终DigiNotar的根证书被所有的浏览器移除，作为一种短期的处理方法，他们的OCSP响应程序被配置成了向所有他们的证书返回“已吊销”。

- **隐私**

  CRL和OCSP都遭受到隐私问题的困扰：当你向一个CA获取吊销信息的时候，你向它暴露了关于你的浏览习惯的一些信息。在CRL情况下泄露信息的可能性比较小，因为CRL通常都含有大量的证书。

  对于OCSP，隐私问题是真实存在的，导致了很多其他的问题。如果一个强有力的攻击者希望观察所有人的浏览习惯，将很容易地通过对一些或者几个主流的OCSP响应程序的观察来实现，而不需要窃听整个世界的真正流量。

  为了解决这个问题，网站应该部署OCSP stapling，这是一种让网站可以响应OCSP的机制。使用这种方法，用户不再需要与CA交互，也就没有了信息的泄露。

5.10.3　证书吊销列表

在开始的时候，*证书吊销列表*（certificate revocation list，CRL）是唯一用于吊销检查的机制。其原理是，每个CA都制作一个吊销了的证书列表，这列表被允许从CA证书中的指定位置下载。客户端应该在信任一个证书之前来查询正确的吊销列表。这个方法被证明是难以扩展的，最终导致了可以进行实时检查的OCSP的诞生。

1. CRL大小的问题

起初，在被作废的证书数量比较少的时候，CRL看起来是有效的。但是当被吊销的证书的数量变得非常巨大的时候，CRL的大小也会随之增大。根据GoDaddy提供的数据，他们的吊销信息从2007年的158 KB增长到了2013年的41 MB。[①]

根据Netcraft提供的数据，他们跟踪了全世界范围的220个CRL，并发现其中很多的体积都很

① NIST Workshop: Improving Trust in the Online Marketplace，http://csrc.nist.gov/groups/ST/ca-workshop-2013/presentations/Koski_ca-workshop2013.pdf（Ryan Koski，2013年4月10日）。

大。[①]其中最大的是CAcert的CRL（CAcert目前不被很多浏览器信任），它的大小在6 MB左右。之后是几个大的列表，再之后是按照大小降序排列的一个很长的CRL。作为说明，表5-3是一个最大的CRL。

表5-3　前10大CRL（来源：Netcraft，2014年3月13日）

CRL	大小（以KB为单位）
CAcert	6219
TrustCenter (Symantec)	1583
Entrust	1460
VeriSign 1 (Symantec)	1346
VeriSign 2 (Symantec)	744
Comodo 1	450
Comodo 2	366
Thawte (Symantec)	346
GoDaddy	320
Comodo 3	314

GoDaddy的41 MB的CRL没有在表5-3中，是因为他们将CRL分成了很多小的列表。其他的大型CA也会使用多列表，这使得CRL的大小问题变得不太透明：如果你是一个活跃的互联网用户，可能需要很多个CRL，这意味着你将持续性地下载大量的数据。如果这对于桌面用户来说不是一个问题的话，那显然对于移动用户则是不可接受的。即使你不担心流量的消耗，CPU对如此大量的文件的处理也会消耗相当的电力。

> **注意**
> CRL大小的问题可以通过使用增量CRL的方式来解决，这种方式每次只需要获取前一个版本的差异。然而，这个特性虽然在所有的Windows平台上都支持，但是在互联网PKI体系中却很少使用。

2. 客户端对CRL的支持

CRL在客户端从来没有得到过很好的支持。今天，这个问题尤其可怕。

- Chrome默认不检查CRL，但是对于EV证书，如果CRLSet（Chrome私有的证书吊销检测方法）设置了并且OCSP没有提供满意答案的时候，会使用CRL。
- Firefox从来不对非EV证书检查CRL。它有一种机制，可以让用户手动配置CRL，在配置之后，Firefox会以固定的时间间隔来下载CRL。但是这个功能在Firefox 24中被移除了。[②]到了Firefox 28，Firefox再也没有检查CRL的功能，即使是EV证书也是如此。[③]

① CRLs tracked by Netcraft，http://uptime.netcraft.com/perf/reports/performance/CRL（Netcraft，检索于2014年7月15日）。

② No CRL UI as of Firefox 24，https://groups.google.com/forum/#!msg/mozilla.dev.security.policy/ilOoDiCU4JM/-iABltbxlWIJ（Kathleen Wilson，2013年8月）。

③ As of Firefox 28, Firefox will not fetch CRLs during EV certificate validation，https://groups.google.com/forum/#!topic/mozilla.dev.security.policy/85MV81ch2Zo（Brian Smith，2013年12月13日）。

- IE浏览器（以及所有依赖于Windows API的其他应用程序）对于CRL是完全支持的，如果在没有其他更好的吊销信息的情况下，它们会下载并使用CRL。
- Safari会检查吊销信息，并会使用所有可用的手段，但是它会忽略掉错误。在我的OS X 10.9笔记本上，OCSP和CRL都被设置成“最佳尝试”。互联网上有很多研究报告（大多是在2011年，在Comodo和DigiNotar被攻击的前后）提出这些功能在之前都是默认“关闭”的。

3. CRL的更新程度

CRL的大小不是唯一的问题。较长的验证周期造成了一个严重的问题并减弱了CRL的有效性。例如，在2013年5月，Netcraft报告了一个已经被吊销的中间证书仍然在悄无声息地被一个主流网站使用。[①]

有问题的证书不包含OCSP信息，但是它的CRL信息是正确的。为什么会出问题？一部分原因是因为没有客户端对中间证书进行CRL检查，这也反映出了对CRL支持的糟糕状态。然而，即使假设浏览器正确使用了CRL（例如IE浏览器），但是CA圈内允许中间证书存在很长的不合理验证周期也依然是一个问题。下面是从基准需求中引用的相关部分。[②]

> 在至少(i)每12个月(ii)在吊销一个从属证书的24小时之内，CA应当更新并重新签发CRL，并且`nextUpdate`**字段的值禁止超过**`thisUpdate`**字段的值12个月**……

因此，中间证书的CRL会被当成在12个月之内是有效的，然而一个严重的吊销可能随时发生在一年中的任何一天。允许如此长的周期可能是为了满足尽可能长的缓存CRL的目的，因为中间证书经常被上百万的网站使用。此外，CRL是被根证书的私钥签名的，这个私钥是放在线下以确保安全的，频繁重新签发CRL可能会影响其安全性。CRL的长更新周期同样对吊销的有效性存在负面影响。这对于中间证书来说尤其明显，如果泄露，中间证书可能用来伪装任何网站。作为对比，服务器证书CRL的更新周期最长为10天。

5.10.4　在线证书状态协议

在线证书状态协议（online certificate status protocol，OCSP）在CRL之后诞生，用来提供实时的证书吊销信息查询功能。OCSP的出现使得在不下载很大CRL的情况下，就可以为每次网站访问提供证书状态查询。

1. OCSP重放攻击

在密码学中，有一个众所周知的针对安全通信的攻击叫作*重放攻击*（replay attack），这种攻击是指攻击者会捕获并利用正确的请求，并很可能是用在不同的上下文中。OCSP按照其最初设计[③]，是不会受到重放攻击的威胁的，客户端会被要求在每次请求的时候提交一个一次性的nonce，

---

① How certificate revocation (doesn't) work in practice，http://news.netcraft.com/archives/2013/05/13/how-certificate-revocation-doesnt-work-in-practice.html（Netcraft，2013年5月13日）。

② Baseline Requirements，https://cabforum.org/baseline-requirements/（CAB论坛，检索于2014年7月13日）。

③ RFC 6960: X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP，https://tools.ietf.org/html/rfc6960（S. Santesson等，2013年6月）。

服务器会在它们签名过的响应中携带同样的值。攻击者无法重放响应，因为每次的nonce都不同。

这个默认安全的方法由于很难推广而终结，在某种意义上，这给了另外一种轻量级、不是那么安全、却很容易在大容量系统中使用的方法提供了推广的机会。轻量级OCSP模式（lightweight OCSP profile）[①]中引入了一系列关于允许批量生成和缓存OCSP响应的建议。为了支持缓存，重放攻击的保护手段需要去除。在没有nonce的情况下，一个OCSP响应就是一个可随意生成、保存并使用CDN交付的普通文件。

这样一来，客户端就不需要在请求中使用nonce。如果客户端使用了nonce（你可以使用OpenSSL的命令行客户端来模拟这种情况），服务器通常会忽略掉它们。因此，对重放攻击的唯一防御手段就是内在的时间限制：攻击者只能在OCSP还没有过期的时候使用它们。这个机会窗口将依赖CA和具体的证书类型（例如，EV证书的响应会有较短的生存周期，但是DV证书的响应生存周期则要长得多），但是这区别很大，从几小时到数天。有效期达到一周的OCSP响应不是很常见。

就像CRL的情况一样，基准需求允许OCSP响应的最大有效期为10天，对于中间证书是12个月。

2. OCSP响应压制

OCSP响应压制（OCSP response suppression）攻击是基于大部分使用OCSP的浏览器都会忽略掉失败这种情况而展开攻击的。这些浏览器诚心诚意地提交OCSP请求，但是当出错的时候则会继续前进而忽略掉错误。因此，一个主动攻击者可以压制吊销检查，这可以通过阻断所有OCSP请求来实现。最简单的方法，就是阻断掉所有与OCSP响应程序之间的连接。也可以伪装成OCSP响应程序并返回HTTP错误。Adam Langley对此进行过尝试，并得出“吊销证书没用”的结论。[②]

在Adam的实验之前，Moxie Marlinspike在2009年发现了一个OCSP协议的缺陷，该缺陷允许不使用网络错误来达到压制OCSP响应的目的。在OCSP中，成功的OCSP响应是进行了数字签名的，这意味着即使是主动攻击者也无法篡改内容。然而，OCSP中存在着几种未经验证的、与错误处理相关的响应类型。如果你的全部需求就是制造一个响应失败，可以简单地返回一个未经验证的错误代码。[③]

3. 客户端OCSP支持

在很多场景下，完全没有必要对OCSP展开攻击，因为客户端会完全忽略OCSP。老的平台和浏览器不使用或者不默认使用OCSP。例如Windows XP和10.7之前版本的OS X就是属于这种。

然而，更重要的是一些现代浏览器主动选择了不使用OCSP。例如，iOS仅对EV证书使用OCSP

---

① RFC 5019: The Lightweight OCSP Profile for High-Volume Environments，http://tools.ietf.org/html/rfc5019（A. Deacon和R. Hurst，2007年9月）。

② Revocation doesn’t work，https://www.imperialviolet.org/2011/03/18/revocation.html（Adam Langley，2011年3月18日）。

③ Defeating OCSP With The Character‘3’，http://www.thoughtcrime.org/papers/ocsp-attack.pdf（Moxie Marlinspike，2009年7月29日）。

（CRL也可能如此）。[①]Chrome在2012年停止使用OCSP[②]，并将所有标准的证书吊销检测方法替换成了一种名为CRLSet的轻量级私有机制[③]。CRLSet提高了吊销检查的性能（所有检测都是在本地，因此很快），但是降低了安全性，因为它们只覆盖了所有吊销信息的一个子集，该子集主要是CA的证书。私有CA是特别有害的，因为没有办法将它们包含在CRLSet中。在Chrome的最近几个版本中，仅会对EV证书尝试OCSP吊销检查，并且仅仅在CRLSet没有覆盖到签发CA的情况下才会尝试。

在2015年3月，Mozilla引入了OneCRL[④]，这是一种针对中间证书（并可能包括少量分支证书）进行吊销检查的私有机制。关于OneCRL的进一步信息，可以参考Mozilla的wiki。[⑤]

即使在使用了OCSP的情况下，几乎所有的浏览器都采用了*软失败*（soft-fail）。它们发出OCSP请求并且对成功的OCSP响应作出正确的处理，却忽略掉所有的错误。在实际中，这只会保护一小部分场景。就像你之前看到的那样，软失败的处理方法无法防御主动攻击者通过阻断连接来进行攻击。

一般来说，关于吊销检查失败最坏的结果就是一个EV网站将失去它的安全状态，使得所有的EV指示标志从浏览器的用户界面上消失。我不确定是否真的有人会注意到这个情况。即使注意到了，他们又将如何应对呢？

4. 响应程序的可用性和性能

从始至终，OCSP都存在不太可靠的这种名声。早期的问题使得浏览器采用了软失败的处理方法，并且OCSP从没被恢复。当今CA已经把他们的响应程序实现得足够稳定，但是浏览器依然拒绝使用硬失败的方式来赌上他们的名声。

> **注意**
> 多亏了Netcraft，我们才能了解到多个CA的OCSP响应程序的性能。[⑥]

这里有以下三个需要独立考虑的问题。

- 可用性

  OCSP响应程序的可用性是最大的问题。如果你正在运行一个安全网站，此时你的CA响应程序出现了故障，你的网站也会受到牵连。如果浏览器实现了硬失败，那么你的网站将

① CRL and OCSP behavior of iOS / Security.Framework，http://stackoverflow.com/questions/5625642/crl-and-ocsp-behavior-of-ios-security-framework（Stack Overflow，回答于2012年3月2日）。

② Revocation checking and Chrome's CRL，https://www.imperialviolet.org/2012/02/05/crlsets.html（Adam Langley，2012年2月5日）。

③ CRLSet，http://dev.chromium.org/Home/chromium-security/crlsets（Chromium Wiki，检索于2014年7月15日）。

④ Revoking Intermediate Certificates: Introducing OneCRL，https://blog.mozilla.org/security/2015/03/03/revokingintermediate-certificates-introducing-onecrl/（Mozilla Security Blog，2015年3月3日）。

⑤ CA: Revocation Plan，https://wiki.mozilla.org/CA:RevocationPlan（Mozilla Wiki，检索于2014年7月15日）。

⑥ OCSP Uptime，http://uptime.netcraft.com/perf/reports/OCSP（Netcraft，检索于2014年7月15日）。

变得不可用。[①]

即使使用软失败，在OCSP响应程序崩溃的时候，也很容易遇到严重的性能问题。使用OCSP的客户端在尝试连接响应程序的时候，会等待网络超时之后才会停止尝试。这个超时一般设置为几秒。例如，Firefox默认使用3秒超时，在使用硬失败的情况下使用10秒超时。

还有一个与所谓的强制网络门户（captive portal）相关的问题，也就是指用户没有完全访问互联网的权限（因此无法访问一些OCSP响应程序）但是仍然需要验证证书的情况。在实际中，这种情况在需要身份验证的无线网络环境中很常见。虽然强制网络门户可以使用白名单来过滤公共OCSP响应程序，但是大部分的门户都没有这样做。

- 性能

OCSP本质上比较慢。它要求客户端首先解析证书，获取OCSP URL，向OCSP响应程序发起一个新的TCP连接，等待响应，然后才能向原始网站发起HTTP请求。一个缓慢的OCSP响应程序可能会让第一次连接你的网站的用户多增加几百毫秒的延迟。

OCSP响应程序的性能可能是CA之间最大的技术差异。你肯定希望选择一个让你的网站速度减慢最小的CA。这样一来，就需要全球分布的OCSP响应程序网络。某些CA使用他们自己的基础设施，而其他的一些CA则会选择商业CDN服务，就像Akamai和CloudFlare。

维护一个健壮的OCSP响应程序并不是一个简单的任务。Verisign（现在的Symantec）运行着一个高可用的OCSP响应程序服务。他们声称，在2012年中他们每天服务了超过45亿次的OCSP查询。[②]一个最新的报告指出，在2014年这个数值增长到了每天140亿。[③]

- 正确性

一个OCSP响应程序是可靠并且高性能的，并不代表它可以正确地作出响应。有些CA不将他们主数据库中的修改与OCSP响应程序同步。例如，之前我从公开CA处购买了一个证书，并将其部署到我的网站上，然后很快就发现所有的OCSP请求都是失败的。

在联系了CA之后，我了解到新签发的证书信息被更新到OCSP响应程序需要最多40分钟的时间。我的意见是将证书的签发推迟到他们的整个基础设施都准备好，但是他们以“太复杂”为由拒绝了这个意见。

当前，OCSP吊销检查很难调整为使用硬失败策略。CA在开始的时候做得不好，因此当浏览器都使用了软失败之后，他们没有足够的动机来改进。今天，一个更有可能的方案是通过OCSP stapling来解决可用性和性能的问题，在这种情况下，允许服务器只从CA获取一次OCSP响应，并将结果用于自己的证书并返回给客户端。

---

① Certificate revocation and the performance of OCSP，http://news.netcraft.com/archives/2013/04/16/certificate-revocation-and-the-performance-of-ocsp.html（Netcraft，2013年4月16日）。

② 2013 Internet Security Threat Report, Volume 18，http://www.symantec.com/security_response/publications/threatreport.jsp（Symantec，2013年4月）。

③ Three years after Diginotar closed, hackers still trying to use its digital certificates，http://www.cso.com.au/article/540551/three_years_after_diginotar_closed_hackers_still_trying_use_its_digital_certificates/（CSO，2014年3月14日）。

> **注意**
>
> 在过去几年中，我一直将我的Firefox浏览器配置成硬失败（在about:config中，将security.ocsp.require设置为true）。在那段时期，我只遇到过一个CA存在OCSP响应程序问题。有趣的是，这与之前那个需要40分钟进行同步的是同一个CA。

5. 强制OCSP stapling

时至今日，看起来整个世界都放弃了通过与签发证书的CA直接通信的方式来提供证书吊销检查的手段。CRL和OCSP都存在问题，以至于它们被浏览器厂商以及网站运维人员所拒绝使用，这些人永远都追求在一无所失的基础上实现高可用性和快速运维。有一种可能性是一个新的特性，叫作强制OCSP stapling，可以将性能以及安全协调到最佳状态。

此特性的核心是使用OCSP stapling，这样网站就可以在每次TLS握手的时候同时提供证书的有效信息。在写作本书的时候，OCSP stapling在隐私、可用性以及性能方面作出了改进。它没有提供任何安全方面的好处，因为它不是必需的，而且可能会被攻击者删除。然而，如果我们强制使用OCSP stapling，那么一个被窃取的证书只有在攻击者能够同时提供有效OCSP响应的时候才有效。这意味着攻击只能在几天内起作用，而不是几个月或者几年直到证书过期为止。

有几个提案是关于如何实现这个特性的。一种提议是创建一种新的证书类型，只针对OCSP stapling的响应有效，具体来说可以通过TLS的扩展[①]来实现。此外，网站也可以对外通告说它要求OCSP stapling，例如通过一个HTTP响应头。实际上，最好的方法可能是通过HSTS的扩展来实现。[②]

这个方法是合理的，因为新的功能能够以扩展的方式加到现有的HSTS结构上。截至目前，Mozilla看起来有兴趣同时支持两种方法。[③]

---

① X.509v3 TLS Feature Extension，https://datatracker.ietf.org/doc/draft-hallambaker-tlsfeature/（P. Hallam-Baker，2015年7月）。

② Requiring OCSP Stapling as a directive in HSTS，https://www.ietf.org/mail-archive/web/websec/current/msg02297.html（Tom Ritter发表于IETF WebSec WG 邮件列表，2015年1月19日）。

③ Bug 901698-implement OCSP-must-staple (off by default)，https://bugzilla.mozilla.org/show_bug.cgi?id=901698（Bugzilla@Mozilla，检索于2014年7月15日）。

第 6 章

实现问题

6

由于种种原因，我们今天所编写的软件天生就是不安全的。第一，那些基本工具（编程语言和库）在编写的时候没有考虑到安全问题。像C和C++这类语言允许我们开发高效但是脆弱的程序。经常可以看到一处编程错误就使得整个程序崩溃。这显然是不合理的。而且在设计库和API的时候也很少考虑到减少错误和提高安全性，很多文档和书籍中的代码和设计都存在基本的安全问题。我们很容易就可以找到一个典型的例子：当下最流行的SSL/TLS库OpenSSL本身就以缺少文档闻名，极难使用。

第二个问题隐藏得更深，它与软件编写背后的经济学原因有关。当今业界奉行的观点是强调用最低的成本（包括时间和金钱）来完成工作，而不会考虑不安全的代码带来的长期影响。安全性（更加通用的说法是代码质量）并不被软件的最终用户所关注，因此软件公司不会在这些方面进行投入。

结果就是，你可能会经常听说绕过而非破解密码学功能的说法。主流的加密基元已经被充分了解，没有攻击者会首先对这些基元进行攻击。但是加密基元很少独自使用，更多的是实现到代码中，与更高层次的协议捆绑在一起使用。这些实现才是导致问题产生的重点，这也是为什么你很少听说有傻瓜来实现他们自己的加密基元库。

尽管在过去发现了很多有关主流加密协议的设计缺陷，但是实现上的缺陷所带来的问题也不容忽视，尤其是那些在没有专业的密码学知识的情况下开发出来的项目，情况更加不容乐观。

本章将回顾主流的实现缺陷，包括过往的和现在还在起作用的那些缺陷。

6.1 证书校验缺陷

为了确保一条TLS连接是可信的，每个客户端都必须执行两个基本检查：检查证书属于预期的域名以及检查证书在有效期内且可信。这听起来很简单，但是细节会带来大问题。开发人员在开发校验证书的代码时，会使用真实的证书来测试代码功能，然而这些真实的证书不会被恶意修改或者引起安全问题，所以开发人员可能会错过编写一些严格测试的逻辑。

例如，下面是一些为校验证书链有效性而需要执行的步骤。（绝非全部！）

(1) 最终实体（服务器）证书对期望连接的域名是可用的。

(2) 证书链中的所有证书（包括最终实体证书）必须进行以下两项检查。

- ❑ 没有过期
- ❑ 签名是有效的

(3) 一个中间证书需要满足以下三个条件。

- ❑ 可以在特定目的下给其他证书签名（例如一个中间证书可以签发Web服务器证书，但是不能给代码签名）
- ❑ 可以签发其他CA证书[①]
- ❑ 可以签发含有域名的最终实体证书

此外，可靠的实现还会检查其他内容，例如，确保所有的密钥都是高强度的，没有使用安全强度低的签名算法（例如MD2、MD5和SHA1）。

6.1.1 在库和平台中的证书校验缺陷

因为众多的软件依赖基础的库和平台，虽然库中的证书校验缺陷并不常见，但是影响依然十分巨大。知名的证书校验缺陷有以下这些。

- ❑ Microsoft CryptoAPI基本约束检查缺陷（2002）[②]

  这可能是大规模使用的库所暴露出来的证书校验缺陷的较早例子，该缺陷影响了所有Microsoft平台以及在其上运行的相关软件产品。该缺陷可以导致任意有效的服务器证书被用于签发一个可信任的欺诈证书，然后使用此欺诈证书进行主动中间人攻击。Konqueror（KDE的默认浏览器）中也发现了同样的问题。之后人们在Microsoft的其他产品中发现了这一缺陷的变种形式，其中某些缺陷可以导致在Windows平台上对代码进行签名。

  这个问题最早由Moxie Markinspike在2002年8月发现。[③]为了演示此问题，Moxie编写了用于模拟中间人攻击的`sslsniff`[④]工具。2009年，Moxie发现在OpenSSL（0.9.6版本前后）中存在着同样的问题，但是没有披露进一步的细节。

- ❑ GnuTLS证书链校验缺陷（2008）[⑤]

  由于证书链校验的代码缺陷，攻击者可以通过在一条不可信的证书链末端加上一张可信根证书的方式达到使证书链被识别成合法的目的。出现这个问题是因为最后增加的可信证书在对证书链中的其他证书进行校验之前就被删除了。

- ❑ OpenSSL中DSA和ECDSA签名校验缺陷（2009）[⑥]

---

① 出于安全原因，签发最终实体证书的CA证书不允许再签发从属CA证书。证书链中所有其他中间证书必须拥有此项权限。

② Certificate Validation Flaw Could Enable Identity Spoofing，https://technet.microsoft.com/library/security/ms02-050（Microsoft Security Bulletin MS02-050，2002年9月4日）。

③ Internet Explorer SSL Vulnerability，http://www.thoughtcrime.org/ie-ssl-chain.txt（Moxie Marlinspike，2002年8月8日）。

④ sslsniff，http://www.thoughtcrime.org/software/sslsniff/（Moxie Marlinspike，检索于2014年2月20日）。

⑤ Analysis of vulnerability GNUTLS-SA-2008-3 CVE-2008-4989，http://article.gmane.org/gmane.comp.encryption.gpg.gnutls.devel/3217（Martin von Gagern，2008年11月10日）。

⑥ Incorrect checks for malformed signatures，https://www.openssl.org/news/secadv/20090107.txt（OpenSSL，2009年1月7日）。

2009年，Google Security Team发现由于OpenSSL代码中存在不足的容错检查，DSA和ECDSA的错误签名不会被检查到。此缺陷的影响是任何进行中间人攻击的攻击者可以使用伪造的证书链，而不会被检查出问题。

- iOS基本约束检查缺陷（2011）[①]

  在差不多10年之后，Apple爆出了一个与Microsoft当年相同的证书链校验缺陷。4.2.10和4.3.5之前版本的iOS系统没有检查证书是否可以作为中间CA使用，从而导致攻击者可以用任意证书签发新的证书。

- iOS和OSX中的连接验证缺陷（2014）

  2014年2月21日，Apple发布了iOS 6.x和7.x的软件更新，这次更新修正了TLS连接验证中的bug。[②]虽然Apple没有提供任何关于这个bug的细节（他们一向如此），但是更新中对于bug的描述吸引了很多人对此bug展开逆向查找，他们发现该bug可能是因为TLS连接验证的代码中的问题导致任何DHE以及ECDHE连接会被主动中间人攻击者所劫持。[③]在最新发布的OSX（10.9，2013年10月发布）中，也发现了同样的bug。不幸的是，Apple并没有针对OSX发布更新以修复此问题。对于这样的严重安全问题，我们尚不清楚Apple为何没有与iOS同步发布针对OSX的软件更新。之后可能是迫于外界压力，Apple在几天后的2月25日发布了含有修改此bug的OSX版本（10.9.2版本）。

  从TLS握手过程的角度来看，这个bug简直严重到了极点，因为它发生在握手过程中的临时部分，因此从不会被应用程序记录相关日志（如果是普通的证书验证攻击，攻击者需要提供伪造的证书链，这些伪造的证书在握手的过程中会被记录到日志中）。所以如果攻击者经过精心设计，只对存在缺陷的客户端发起攻击（鉴于TLS握手暴露了足够的信息从而允许非常可靠的指纹识别，攻击应该有可能实现），那么就可以将攻击做到可靠、高效、悄无声息而不留下任何痕迹。

  在这些有缺陷的系统上运行的所有应用程序都受到影响。唯一的例外是那些跨平台的应用程序（例如Chrome和Firefox），这些程序依赖它们自由的TLS库。

- GnuTLS证书链校验缺陷（2014）

  2014年早期，GnuTLS披露了两个与证书链校验有关的漏洞。[④]第一个漏洞会导致GnuTLS将任意版本号为1的X.509格式的证书识别为中间CA证书。如果攻击者想方设法获取到了一个有效的v1版本的服务器证书（一般来讲不太可能，因为这个版本已经废弃了），他们就可以使用这个证书来伪装成任何的服务器以欺骗使用GnuTLS的客户端。这个漏洞是在GnuTLS 2.11.5中引入的。

  再来看第二个漏洞。在Apple的TLS验证问题被发现之后不久，GnuTLS也披露了一个类似的

---

① TWSL2011-007: iOS SSL Implementation Does Not Validate Certificate Chain，http://seclists.org/fulldisclosure/2011/Jul/315（Trustwave SpiderLabs，2011年6月25日）。

② About the security content of iOS 7.0.6，https://support.apple.com/en-gb/HT202934（Apple，2014年2月）。

③ Apple's SSL/TLS bug，https://www.imperialviolet.org/2014/02/22/applebug.html（Adam Langley，2014年2月22日）。

④ Advisories，http://gnutls.org/security.html（GnuTLS，检索于2014年7月17日）。

bug：使用一个篡改过的证书可以导致GnuTLS的验证过程缩短并将此证书判断为有效。[①]这估计是GnuTLS的开发者在了解到Apple的bug之后，重新检查了自己的代码而发现的。虽然GnuTLS没有被主流的浏览器使用，在服务器端也不像OpenSSL那样流行，但是它依然有一些主要的用户。例如，很多Debian发行版中自带的软件包使用的就是GnuTLS。因此此缺陷造成的影响可能非常巨大。这个bug在GnuTLS的代码中存在了相当长的时间，极有可能从GnuTLS的第一个版本开始就存在了。

- OpenSSL的ChangeCipherSpec消息注入缺陷（2014）

  2014年6月，OpenSSL开源项目公布了一个长期存在的缺陷，该缺陷使得主动网络攻击者可以在SSL/TLS握手的过程中通过注入ChangeCipherSpec消息的方法，迫使最终协商出预期的主密钥值。[②]此问题几乎存在于所有版本的OpenSSL中，但是目前据我们所知，除了使用特定有缺陷版本的1.0.1分支的OpenSSL的服务器以外，该缺陷还不能对其他的版本造成危险。造成此问题的根本原因是：在TLS握手的过程中，ChangeCipherSpec消息是握手双方用来通知对方协商过程结束并进入加密阶段，但是此消息并没有经过验证，因为它不属于TLS握手协议的一部分。如果攻击者提前发送了这个消息（OpenSSL对此应该发现异常但是实际上并没有），那么存在威胁的一方会使用攻击者了解的信息过早地生成密钥。[③]尽管这个缺陷十分严重并且容易被利用，但是它的影响却不是很大，因为要利用这个缺陷，需要通信双方都使用OpenSSL，然而OpenSSL在客户端很少有人使用。在将OpenSSL用于客户端应用程序的平台中，最著名的是Android 4.4（KitKat）平台，该平台随后修复了这个问题。据SSL Pulse的数据，就在该缺陷刚刚公布的时候，全球大概有14%的服务器在运行存在此缺陷版本的OpenSSL。

- OpenSSL可选链证书伪造（2015）

  在2015年7月发现了OpenSSL的另外一个身份验证漏洞。在证书链校验代码中的一个bug导致网络攻击者可以使用分支证书伪造成有效的中间证书（CA）。[④]幸运的是，这个问题的影响很小，因为该缺陷只是在一个月前引入的，绝大多数服务器都不会如此紧密地跟随OpenSSL的升级。基于一个针对此问题的测试用例的公布，[⑤]Metasploit中增加了针对此问题的攻击模块[⑥]。

2014年，一组研究人员公布了对几个SSL库所进行的针对证书校验缺陷的综合对抗性测试的

---

① Dissecting the GnuTLS Bug，http://blog.bro.org/2014/03/dissecting-gnutls-bug.html（Johanna，2014年3月5日）。

② OpenSSL Security Advisory CVE-2014-0224，https://www.openssl.org/news/secadv/20140605.txt（OpenSSL，2014年6月5日）。

③ Early ChangeCipherSpec Attack，https://www.imperialviolet.org/2014/06/05/earlyccs.html（Adam Langley，2014年6月5日）。

④ Alternative chains certificate forgery (CVE-2015-1793)，https://openssl.org/news/secadv/20150709.txt（OpenSSL Security Advisory，2015年7月9日）。

⑤ Add test for CVE-2015-1793，https://github.com/openssl/openssl/commit/593e9c638c58e1a510c519db0d024527113330f3（OpenSSL commit，2015年7月2日）。

⑥ Add openssl_altchainsforgery_mitm_proxy.rb，https://github.com/rapid7/metasploit-framework/pull/5735（Metasploit pull request，2015年7月16日）。

结果。[①]他们提出了一个叫作“突变证书”的概念来伪造证书，即frankencerts。这些伪造的证书是基于真实的证书创建的。[②]测试结果显示，虽然目前被广泛使用的SSL库和主流浏览器都通过了测试，但是那些较少被使用的库（例如PolarSSL、GnuTLS、CyaSSL以及MatrixSSL等）都存在严重缺陷。

6.1.2 应用程序校验缺陷

如果说主流的平台和库中都存在着严重的证书校验缺陷，那我们有理由认为其他软件的情况将更加糟糕。毕竟，对于大多数开发人员来说，过多思考代码的安全性会影响他们及时交付软件给用户。业内有很多关于最终用户程序代码中的证书校验逻辑缺陷的传闻，并存在相关的证据，但是直到2012年一篇关于该主题的研究报告发表后，人们才清楚地了解此问题的情况。[③]以下是从报告中摘录的重点：

> 我们证实了在很多安全关键的应用程序和库中都存在SSL证书校验逻辑的缺陷。有问题的软件包括Amazon EC2的Java库以及所有基于其上的云客户端；Amazon和Paypal的负责将支付详情从电商网站发送到支付网关的零售SDK；osCommerce、ZenCart、Ubercart和PrestaShop这些一体化购物车软件；移动端网站使用的AdMob代码；Chase移动银行与几个其他的Android APP和库；包括Apache Axis、Axis 2、Codehaus XFire和Pusher for Android在内的Java Web服务中间件，以及所有使用这些中间件的应用程序。**任何来自这些程序的SSL连接在中间人攻击的面前都是不安全的**。

如果这些都无法敲响警钟，那我也不知道什么会了。很明显，报告中提到了一些主要的互联网基础设施的组件。按照编写该报告的研究团队的观点，造成这个情况的主要原因是不好的API设计。很多库不仅是在默认状态下很不安全（没有任何证书校验），而且想使用它们写出安全的代码也很困难。大部分库本身提供的功能过于底层，这就更多地依赖用户来确保安全性。例如OpenSSL就只能由使用人员自己编写代码来执行主机名验证。

这个报告非常准确地描述了我们整个开发工具栈中的一个主要问题，包括全部的代码和安全性，不仅仅是SSL和TLS。没错，有很多的库并不安全也难以使用，但是真正的问题在于我们居然一直在使用它们。难怪我们会一直犯同样的错误。

公平地讲，还是有一些平台做得不错。例如Java的SSL/TLS实现（JSSE）默认执行了所有必要的校验，这显然很对那些不希望自己搭建安全开发组件的程序员的胃口。坊间证据表明，大多数程序员在开发阶段会禁用代码中的校验逻辑。我们怀疑这些校验逻辑在生产环境中会有多少会被再次启用。

---

① Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations，https://www.cs.utexas.edu/%7Eshmat/shmat_oak14.pdf（Brubaker等，S&P，2014年）。

② Frankencert，https://github.com/sumanj/frankencert（sumanj，GitHub，检索于2014年7月17日）。

③ The most dangerous code in the world: validating SSL certificates in non-browser software，https://crypto.stanford.edu/~dabo/pubs/abstracts/ssl-client-bugs.html（Georgiev等，CCS，2012年）。

6.1.3　主机名校验问题

说到主机名校验，去校验一个证书是否符合期望的主机名能有什么难的？但结果是，这种校验经常被错误地实现，有一些漏洞说明了这一点。在2009年的美国黑帽子（Black Hat USA）大会上，Dan Kaminsky[1]和Moxie Marlinspike[2]各自演示了如何通过证书校验缺陷实施中间人攻击，而被劫持的受害者没有收到任何关于证书失效的警告。

要想成功地实施完整攻击需要同时利用几个缺陷，但是他们两人的方案中的共同点是利用了NUL字节，即C和C++中用做字符串结尾标志的0字节。在C和C++这种场景里，NUL字节不是数据的一部分，而是指明数据结束。这种表示文本数据的方法是方便的，因为你需要记录一个指向数据的指针。在处理数据的时候，如果遇到了NUL字节，那么你就会知道到达了数据的结尾（如图6-1所示）。

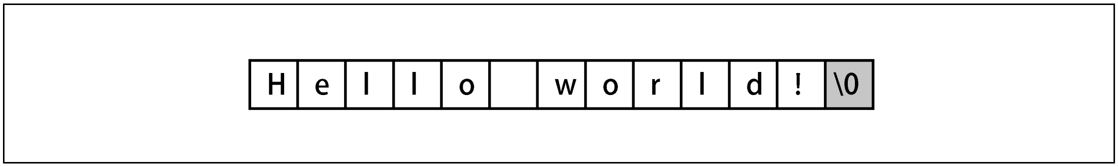


图6-1　C字符串在内存中的表示

基于ASN.1编码标准的证书结构使用了一种不同的方法，这种方法中，所有的结构都是带长度存储的。处理字符串长度的不同方法导致了问题的发生：证书使用一种方式编码（ASN.1），但是使用另一种方式处理（C字符串）。

那么针对此场景的攻击是这样的：构造一个在主机名中包含NUL字节的证书，然后打赌：(1) 大部分客户端会判定NUL字节处就是主机名的结尾；(2) 主机名中的NUL字节可以混过CA的校验流程。

以下是Moxie执行攻击的步骤。

(1) 创建一个含有NUL字节的特殊主机名称字符串。Moxie使用的是：www.paypal.com\0.thoughtcrime.org（NUL字节使用\0表示，但正常情况下是“看不见”的；如图6-2所示）。构建字符串的规则是下面两个。

- 将你想仿冒的主机名放在NUL字节的前面。
- 将你能控制的一个域名放在NUL字节的后面。

(2) 对于CA来说，NUL字节并不是什么特殊的字符。[3]他们签发证书的时候是基于域名的结尾

---

[1] PKI Layer Cake: New Collision Attacks Against the Global X.509 Infrastructure，https://www.cosic.esat.kuleuven.be/publications/article-1432.pdf（Kaminsky等，Black Hat USA，2009年）。

[2] More Tricks For Defeating SSL In Practice，http://www.blackhat.com/presentations/bh-usa-09/MARLINSPIKE/ BHUSA09-Marlinspike-DefeatSSL-SLIDES.pdf（Moxie Marlinspike，Black Hat USA，2009年）。

[3] 实际上，从严格意义上并不能这么说。某些CA被发现不正确地处理NUL字节，并且将它误以为是字符串结束符。在当今，几乎所有CA都对提交的域名执行了各种形式的检查。

进行校验，这会导致校验流程命中某个顶级域名。在本例中，CA需要校验的域名是thoughtcrime.org，该域名属于Moxie。那他当然可以证明证书是他申请的。

(3) 最后这个证书就可以放在一个修改过的`sslsniff`中，用来针对客户端展开攻击。

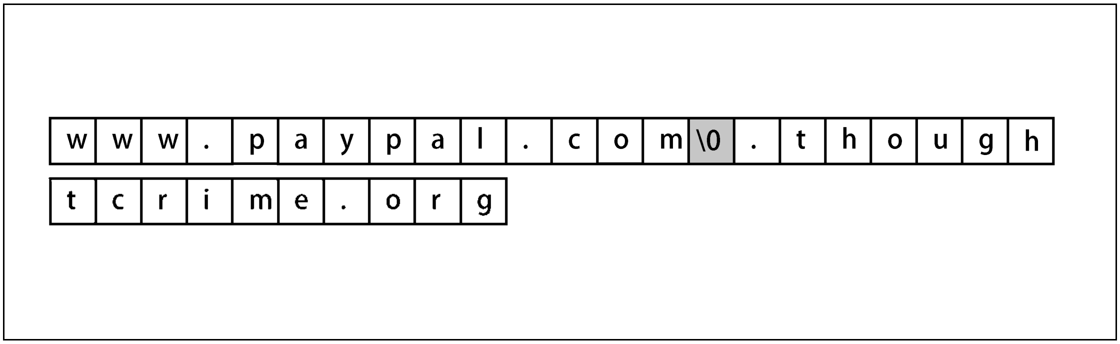


图6-2　Moxie Marlinspike使用的用于演示攻击的域名

Microsoft的CryptoAPI、GnuTLS和NSS库中都发现了NUL字节攻击的漏洞，这影响了Firefox、Internet Explorer和许多其他浏览器。此外如果你使用泛域名的形式来签发一个形如*\0thoughtcrime.org的证书，就会得到一个万能的中间人攻击证书。

6.2　随机数生成

由于所有的密码学算法都依赖随机数，生成随机数的功能就成为了建立安全信道的关键部分。[①]例如，在生成一个密钥的时候，你就需要生成随机数。需要注意的是，这里说的生成密钥不是那种一次性的行为（例如在一台新服务器上安装密钥），而是在类似TLS这种安全协议中为每个连接所生成的密钥。

举例来说，如果使用一个好的随机数生成器（random number generator，RNG），一个256位的对称密钥可以提供256位的安全强度（与强算法一起使用）。但是如果随机数生成器存在缺陷，则你不会在256位的这个大空间中得到一个随机值，相反，你可能只会在相比小得多的32位的空间中生成一个随机数。随机数的有效空间越小，安全性就越差。如果密钥的有效位数太小，那么即使使用暴力破解的方式也可以获得密钥。

6.2.1　Netscape Navigator浏览器（1994）

早期的一个随机数生成缺陷的例子来自Netscape Navigator浏览器，这款当时的旗舰产品诞生于设计了SSL协议的公司。这款浏览器使用了一个简单的随机数生成算法，这个算法依赖系统启

① 真正的随机数生成是不可能的，除非使用特殊的硬件。在实际中，我们依赖伪随机数生成器（pseudorandom number generator，PRNG）。大部分PRNG使用少量熵作为种子，然后就可以生成大量的伪随机数。在本节中，我交替使用RNG和PRNG。

动到当前的时间（以毫秒为单位）和底层操作系统进程以及其父进程的进程ID。该问题在1995年的时候被披露出来，起因是两个研究人员对Netscape Navigator的随机数生成器[1]进行了逆向工程后，编写了一个程序，成功地破解了主加密密钥[2]。

对于攻击者来说，最理想的情况就是在目标Unix机器上也拥有一个账号，这样他就可以获得进程和父进程的进程ID。攻击者可以通过网络上的数据包获得系统的秒级时间，那么问题就变成了猜测毫秒，而这只有20位的安全强度。要获得这20位的毫秒数，攻击者只需要使用普通的硬件，就可以在25秒的时间内完成破解。

更现实的情况下，攻击者不知道进程ID，那问题规模会被减小到47位，这依然可以使用暴力破解的方法在短时间内达成。

6.2.2　Debian（2006）

2008年5月，Luciano Bello发现Debian Project在2006年9月的一个灾难性的编程错误影响了OpenSSL系统库中使用的随机数生成器。[3]这个bug在2007年4月发布的Debian Etch稳定版中得到了修正。Debian不仅是一个被广泛使用的Linux发行版，同时也是很多其他发行版的基础版本（最著名的是Ubuntu），这也说明了这个问题确实影响了大量服务器。

这个编程错误本质上是一行代码被偶然移除了（注释掉），该行代码的作用是输入熵（entropy）给随机数生成器。该行代码移除之后，随机数生成器的熵就只剩下一些从进程ID得来的辅助性数值，这意味着对于所有的密码学操作都只有16位的熵。在这种情况下，所有受影响的系统的安全性都将不复存在。

产生问题的代码片段如下所示：[4]

```
/*
* 请勿添加未初始化的数据
        MD_Update(&m,buf,j);
*/
        MD_Update(&m,(unsigned char *)&(md_c[0]),sizeof(md_c));
        MD_Final(&m,local_md);
        md_c[1]++;
```

在实际应用中此错误产生的最大问题是弱OpenSSH密钥[5]，但是由于SSH密钥保存的位置都是广为人知的，可以很容易地进行检查，此问题从而得到了很好的缓解。Debian项目构建了一个

① Randomness and the Netscape Browser，http://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html（Ian Goldberg和David Wagner，1996年1月）。

② unssl.c，http://www.mavi1.org/web_security/cryptography/General/unssl.c（Ian Goldberg和David Wagner，1995年9月）。

③ DSA-1571-1: openssl — predictable random number generator，http://www.debian.org/security/2008/dsa-1571（Debian，2008年5月13日）。

④ Diff of /openssl/trunk/rand/md_rand.c r140:r141，http://anonscm.debian.org/viewvc/pkg-openssl/openssl/trunk/rand/md_rand.c?r1=141&r2=140&view=diff&pathrev=141（Debian OpenSSL package, 2006年5月2日）。

⑤ Working exploit for Debian generated SSH Keys，http://seclists.org/fulldisclosure/2008/May/410（Markus Müller，2008年5月15日）。

有风险密钥的黑名单，并提供了检测这些密钥的工具。

替换掉有风险的TLS密钥则困难得多，因为替换行为无法作为自动升级过程的一部分来完成。人们开发了很多脚本来检查所有的文件以及检测弱密钥。因为该问题可以通过检查服务器的公钥来确定，所以可以有效地使用远程工具进行检测，例如我在SSL Labs网站上就提供了一个这样的工具。此外，多数服务器证书的有效期在1~2年的范围内，CA可以检查公钥（公钥位于证书的签名请求中）然后拒绝为有风险的公钥签发证书。然而即便如此，人们依然对这个问题充满了疑惑。许多人发现即使在有风险的系统上生成了一些密钥，这些密钥还是无法被检测工具有效地识别为有风险的密钥。

Debian随机数生成器问题的发现，揭示了开源项目的代码经常被一些不熟悉代码逻辑的人随意修改（无论基于什么原因）的事实。即使像OpenSSL这样关键的系统组件也没有得到足够的质量保证。最后，数以百万计的服务仍一直依赖这样的代码。

在开源圈子里，开源项目的开发者和发行版的打包者之间的矛盾早已广为人知。[①]发行版经常创建各种开源项目的分支并且对它们进行大量的修改，但是却继续使用这些项目原有的名称。结果就是，他们经常在哪些版本存在问题以及由谁来负责修复问题这样的事情上产生纠纷。造成这种现象的根本原因是开发者和打包者在开发计划、任务优先级以及开发目标上存在不同而产生冲突。[②]

6

> **注意**
>
> Debian不是唯一遭遇随机数生成问题的操作系统。2007年，三位研究人员发布论文讨论了Windows 2000的随机数生成器的弱点。[③]晚些时候，人们在Windows XP中也发现了同样的问题。2013年3月，NetBSD项目宣布，最初于2012年10月发布的NetBSD 6.0在内核的随机数生成器中存在bug，影响系统的安全强度。[④]

6.2.3 嵌入式设备熵不足问题

2012年2月，一组研究人员发表了关于互联网上RSA和DSA密钥质量的延展研究的结果。[⑤]结论显示，观察到的RSA密钥（在SSL/TLS协议中使用）至少有0.5%是不安全的，而且容易被攻破。DSA密钥（在SSH中使用）的结果更糟，有1.03%的密钥是不安全的。

造成密钥不安全的大多数原因都是随机数生成的问题。下面是研究的结论。

> 最后，我们的研究结论应该引起人们警醒：在实际应用的重要领域里，安全随机数的生成依然是一个没有解决的问题。

积极的一面是，几乎所有问题都出现在无头系统和嵌入式设备中，研究结果显示几乎所有的

---

① Vendors Are Bad For Security，http://www.links.org/?p=327（Ben Laurie，2008年5月13日）。

② Debian and OpenSSL: The Aftermath，http://www.links.org/?p=328（Ben Laurie，2008年5月14日）。

③ CryptGenRandom，https://en.wikipedia.org/wiki/CryptGenRandom（维基百科，检索于2014年7月17日）。

④ RNG Bug May Result in Weak Cryptographic Keys，http://ftp.netbsd.org/pub/NetBSD/security/advisories/NetBSD-SA2013-003.txt.asc（NetBSD，2013年3月29日）。

⑤ Widespread Weak Keys in Network Devices，https://factorable.net/（factorable.net，检索于2014年7月17日）。

非嵌入式系统中的密钥都是安全的。在存在问题的证书中，只有少量是由公开CA所签发的。识别出的主要问题包括以下这些。

- 默认密钥

  一些厂商在交付产品的时候会内置一些默认的密钥。显然，这种行为没有达到厂商期望的目的，因为所有产品的用户最终都会使用同样的密钥，并且用户之间可以互相攻破对方，只要他们从产品的软件或者硬件提取出私钥。此外，这些密钥也将不可避免地在全世界扩散。[①]

- 低熵导致的重复密钥

  一些设备在第一次启动的时候生成密钥，此时用于生成随机数的熵是不足的。这种方式生成的密钥几乎都是可以预测的。研究报告描述了一组模拟无头系统初次启动的Linux的实验，实验清楚地演示了Linux内核的熵收集系统在初次启动的最初几秒中存在的弱点。

- 可因子分解的密钥

  最有意思的是，对于RSA密钥，研究发现很多密钥中用于计算模数的两个素数中的一个是相同的，这使得密钥可能会被攻破。原则上素数应该是随机生成的，生成相同的素数是不该发生的。研究指出，发生这种问题的根本原因是OpenSSL生成RSA密钥的代码逻辑在低熵条件下工作不正常。

与TLS相关的研究结果的汇总如表6-1所示。

表6-1　有风险的私钥汇总（来源：factorable.net）

活动主机的数量	12 828 613	(100.00%)
使用重复密钥	7 770 232	(60.50%)
使用有风险的重复密钥	714 243	(5.57%)
使用默认证书或默认私钥	670 391	(5.23%)
使用低熵重复密钥	43 852	(0.34%)
使用可以分解因子的RSA密钥	64 081	(0.50%)
使用Debian弱密钥	4147	(0.03%)
使用512位的RSA密钥	123 038	(0.96%)
被标识成有风险的设备型号	985 031	(7.68%)
使用低熵重复密钥	314 640	(2.45%)

很明显，在各个层面都存在缺陷（例如，厂商可以检查这些问题并规避），但是该研究最终揭露的是一个真正的可用性问题：位于底层操作系统之上的提供密钥功能的应用程序期望底层系统提供足够的随机度，但系统提供的随机度往往不够。当随机度不够的时候，应用程序也没有办法来直接判断问题的发生（例如，Linux永远不会阻塞对`/dev/urandom`的读操作）。少数应用程序使用纵深防御的方法和基于统计的测试来验证它们得到的随机数是否真的随机。

① LittleBlackBox，https://code.google.com/p/littleblackbox/（Database of private SSL/SSH keys of embedded devices，检索于2014年7月17日）。

无法依赖操作系统提供可靠随机度的问题迫使一部分开发者转而实现自己的随机数生成器。然而这种方式很少成功，因为随机数的生成是一件容易出错并且很难的工作。

如果你有一个嵌入式设备并且希望检查这个设备中密钥的质量，该研究的作者们提供了一个在线工具，可以用来检测任何接入互联网的设备。

6.3 心脏出血

心脏出血[①]是OpenSSL中的一个灾难性漏洞，它在2014年4月被发现并公开。此攻击利用了心跳（Heartbeat）协议实现中的一个缺陷，心跳协议是一个很少使用的TLS协议扩展（更多关于心跳的内容请参考2.12.4节）。

心脏出血漏洞可以说是TLS协议有史以来遭遇的最严重的问题，比较讽刺的是，这个漏洞与密码学缺陷并没有什么关系。这个漏洞成为了显示当今开源软件糟糕的开发状态和代码质量的确凿证据。

当心脏出血这颗“核弹”爆炸之后，所有人的目光都被吸引到了OpenSSL上。虽然OpenSSL项目缺少资金支持而且代码质量较差的问题已经暴露了很久，但是OpenSSL社区最终只能依靠这样一个巨大的漏洞来解决这些问题。结果是有好有坏，具体怎么认为取决于你自己的观点。Linux基金会宣布了一项为期三年，名为Core Infrastructure Initiative（核心基础设施倡议）的项目，旨在为那些处于资金不足状态的开源项目提供总额为390万美元的资金。[②]在这个项目的帮助下，OpenSSL随后公布了确认和解决产品中所存在问题的路线图。[③]与此同时，OpenBSD社区基于OpenSSL项目创建了自己的LibreSSL分支，开始对其进行快速迭代以期可以改进代码的质量。[④]

6.3.1 影响

因为OpenSSL的代码中没有对读长度进行检查，攻击者可以利用这个缺陷，在一个心跳请求中获取到服务器进程中最大为64 KB的数据。通过发出多个这样的请求，攻击者就可以无限制地获取内存数据。如果在服务器的进程中存在一些敏感信息（这是必然会存在的），攻击者就可以得到这些信息。因为OpenSSL主要的工作是处理加解密，所以攻击者最希望获取到的内容就是服务器的私钥。此外也有其他一些有价值的信息可以被获取：会话票证的密钥、TLS的会话密钥以及各类密码等。

OpenSSL从1.0.1版本到1.0.1f版本都存在心脏出血漏洞；更早期的版本则不受影响，例如0.9.*x*和1.0.0。可想而知，数量巨大的服务器都受到了影响。Netcraft估计，这些服务器的数量占全球

---

① Heartbleed，https://en.wikipedia.org/wiki/Heartbleed（维基百科，检索于2014年5月19日）。

② Tech giants, chastened by Heartbleed, finally agree to fund OpenSSL，http://arstechnica.com/information-technology/2014/04/tech-giants-chastened-by-heartbleed-finally-agree-to-fund-openssl/（Jon Brodkin，Ars Technica，2014年4月24日）。

③ OpenSSL Project Roadmap，https://web.archive.org/web/20150815130657/http://openssl.org/about/roadmap.html（OpenSSL，检索于2014年7月17日）。

④ LibreSSL，http://www.libressl.org/（OpenBSD，检索于2014年7月17日）。

服务器总量的17%左右，从数量上来说，这大概是50万台。[①]

值得注意的是，到目前为止几乎所有受影响的服务器都进行了修补。在问题的严重性、免费获取的检查工具以及媒体的关注等多种因素的共同作用下，这堪称TLS史上速度最快的漏洞修复。一个全互联网扫描组织宣称在心脏出血漏洞爆发之后的一个月后，全网有1.36%的HTTPS服务存在漏洞。[②]而与此同时，SSL Pulse资料显示，Alexa的排名靠前的网站中，仅有0.8%依然存在心脏出血漏洞。

在发现漏洞之后不久，大部分声音都是建议更换私钥以预防私钥泄露，但当时还没有真实的私钥泄露的案例发生。这可能是因为人们起初都忙于测试并修复漏洞。之后，当人们的注意力回到如何利用这个漏洞进行攻击的时候，才明显注意到各种窃取服务器私钥的案例。[③]人们发现在某些情况下，私钥在经过很多次心跳请求后会被窃取，而在其他情况下，只是一些较少的心跳请求就会导致私钥丢失，而更加高级的利用漏洞的攻击方法也随后被发现。[④]

在漏洞被公开之后不久那些日子里，利用此漏洞对网站进行攻击的行为是非常猖獗的。不仅私钥是攻击者窃取的目标，例如，Mandiant报告称，他们检测到一次成功的针对VPN服务器的攻击，这次攻击使得多因子身份验证的机制被绕过。攻击者利用这次攻击，成功地从服务器内存中提取了TLS会话密钥。[⑤]

加拿大税务部门中保存的社保号码被窃取，Mumsnet网站（英国最著名的育儿资讯网站）的用户密码也被黑客获取。[⑥]

现在，有很多公开的工具可以很容易地让任何人利用心脏出血漏洞在几分钟内攻击一台存在漏洞的服务器。其中某些工具十分先进，甚至可以提供全自动化的私钥获取功能。

> **注意**
> 如果你希望了解有关该bug本身的更多信息，并希望了解如何针对易受攻击的服务器进行测试，请参考12.14节。

① Half a million widely trusted websites vulnerable to Heartbleed bug，http://news.netcraft.com/archives/2014/04/08/half-a-million-widely-trusted-websites-vulnerable-to-heartbleed-bug.html（Netcraft，2014年4月8日）。

② 300k servers vulnerable to Heartbleed one month later，http://blog.erratasec.com/2014/05/300k-servers-vulnerable-to-heartbleed.html（Robert Graham，2014年5月8日）。

③ The Results of the CloudFlare Challenge，https://blog.cloudflare.com/the-results-of-the-cloudflare-challenge/（Nick Sullivan，2014年4月11日）。

④ Searching for The Prime Suspect: How Heartbleed Leaked Private Keys，https://blog.cloudflare.com/searching-for-the-prime-suspect-how-heartbleed-leaked-private-keys/（John Graham-Cumming，2014年4月28日）。

⑤ Attackers Exploit the Heartbleed OpenSSL Vulnerability to Circumvent Multi-factor Authentication on VPNs，https://www.mandiant.com/blog/attackers-exploit-heartbleed-openssl-vulnerability-circumvent-multifactor-authentication-vpns/（Christopher Glyer，2014年4月18日）。

⑥ Heartbleed hacks hit Mumsnet and Canada's tax agency，http://www.bbc.co.uk/news/technology-27028101（BBC，2014年4月14日）。

6.3.2 缓解方法

打补丁是处理心脏出血漏洞的最好方法。如果你使用的是系统提供的OpenSSL库，那么系统的提供商一定已经提供了相应的修复。如果你是自己从源代码编译OpenSSL，那么可以重新编译最新的OpenSSL 1.0.1版本。在使用源代码编译的情况下，还可以配置OpenSSL以删除对心跳协议的支持。这可以通过使用OPENSSL_NO_HEARTBEATS标志来实现，例如：

```
$ ./config -DOPENSSL_NO_HEARTBEATS
$ make
```

在这之后，你可能需要重新编译所有依赖OpenSSL的静态软件包。

很多产品（例如一些设备）内嵌了OpenSSL并且很可能存在风险。因为它们没有针对心脏出血给出警告，这些设备在漏洞公布的时候都没有做好修复的准备。拥有很多这类产品的厂商可能正在疲于应对为所有的产品发布修复补丁。

当完成对漏洞本身的修补之后，将你的注意力转到那些可能从服务器泄露的敏感数据上。至少，你需要替换掉服务器的私钥，去重新签发新证书并作废旧的证书。Netcraft观察了全球范围的心脏出血漏洞的修复状态，表示很多网站依然遗漏了若干上述步骤。[①]

在处理完私钥和证书后，再来关注服务器内存中可能存在的其他数据。会话票证的对称密钥很明显是攻击者的另一个目标，需要将它们替换掉。之后，可以考虑一下其他的密码，比如用户密码。根据你的风险预测，可能需要建议或者要求你的用户修改密码，就像一些其他网站已经做过的那样。

心脏出血不能获取访问数据库的权限，至少不能直接获取；但是可以通过心脏出血间接地获取一些有用的信息。例如，在一个数据库驱动的网站中，每个请求都会使用数据库密码并使得密码留在内存中。替换掉所有的内部密码是保持安全的最好方式。

那些在心脏出血漏洞发生前就部署了前向保密的网站则处于最佳的状态：它们的历史加密通信无法被泄露的服务器私钥所解密。如果你不是处于这种状态，那么请现在就部署前向保密。这就是为什么这个特性如此重要的原因。

> **警告**
> 虽然我们一直关注的是服务器端，但是客户端使用有漏洞的OpenSSL版本也存在风险。心跳是一个双向协议。如果一个存在风险的客户端连接到一个恶意服务器，那么这个服务器就可以获取到客户端进程的内存数据。[②]

6.4 FREAK

2015年1月，OpenSSL发布了一个低严重级别的公告，警告说客户端会在一个全安全强度

① Keys left unchanged in many Heartbleed replacement certificates，http://news.netcraft.com/archives/2014/05/09/keys-left-unchanged-in-many-heartbleed-replacement-certificates.html（Netcraft，2014年5月9日）。

② Pacemaker，https://github.com/Lekensteyn/pacemaker（Heartbleed client exploit，检索于2014年5月19日）。

（full-strength）的RSA握手过程中接受使用弱安全强度的出口RSA密钥。其中关键在于客户端并没有允许协商任何出口级别的RSA密码套件。该问题被标识为CVE-2015-0204，没有引起太多的注意。

一些研究者关注到了该公告背后的本质原因，于是开始研究基于此问题的攻击。同年3月初，他们宣布成功地利用CVE-2015-0204对他们自己和www.nsa.gov网站的加密通信进行了一次中间人攻击。同时这些研究人员给这种攻击起了一个名称：FREAK（Factoring RSA Export Keys）。

这次攻击演示的意义在于说明互联网上有大量的服务器受到此问题的威胁而容易遭受中间人攻击。归功于这些研究者的努力，CVE-2015-0204被重新标识为高严重级别。

6.4.1　出口密码

在讨论FREAK之前，我们需要回到20世纪90年代，了解什么是出口强度的密码套件。在1998年9月之前，美国曾经限制出口高强度的加密算法。具体来说，限制对称加密强度为最大40位，限制密钥交换强度为最大512位。出口密码套件就是为符合这种限制而精心设计的。

但是仅仅定义弱强度密码是不够的，为了得到更好的性能，纯RSA握手将身份验证和密钥交换捆绑到一起。虽然出口限制条例允许强身份验证，但是无法将身份验证过程单独从密钥交换过程中分离出来。解决方法是扩展握手协议以便产生与出口密码套件相一致的弱强度的RSA密钥。这样，一个拥有高强度RSA密钥的服务器依然可以用它来进行身份验证。对于密钥交换来说，服务器会生成一个512位的RSA私钥，并在需要使用出口密码套件的时候使用这个私钥。[①]

出口密码套件在2000年1月之后由于美国放松了对密码技术的出口控制而被废弃，但是支持出口密码套件的代码依然存在于为数众多的SSL/TLS库中。就像已经发生的那样，如果恶意修改SSL协议在握手过程中的某些消息，则这些老旧代码可能会被重新触发使用。

6.4.2　攻击

在正常的RSA握手过程中，客户端随机生成预主密钥，并用服务器的公钥加密后发送给服务器。如果RSA的密钥强度足够高，则密钥交换过程也是强安全级别的。当出口密码套件被协商后，服务器生成一个弱强度的RSA公钥，使用高强度的RSA私钥对其签名，并使用`ServerKeyExchange`消息将签名后的弱RSA公钥发给客户端。之后客户端使用这个弱RSA公钥加密预主密钥以符合出口条例。虽然公钥的强度较低，但是攻击者无法随意使用这个公钥进行主动攻击，因为针对公钥的签名的强度依然很高。

> **注意**
> 为了理解本节讨论的全部内容，你需要了解一些SSL/TLS协议的细节。如果你对协议细节还不是很了解，那么在继续之前，请先阅读第2章以了解协议和握手的相关细节。

---

① 另外一个特性叫作服务器网关加密（server-gated cryptography，SGC），其引入是为了仅让选定服务器对那些使用弱加密的客户端使用强加密。主要的思路是在为美国公司签发的证书中加入特殊标记。在发现特殊标记之后，客户端会透明地进行重新协商从而使用强密码策略。

从当今的角度看，出口密码套件的安全强度是非常弱的。如果使用了这种套件，那么密钥交换将基于一个弱RSA密钥来进行。虽然攻击者无法直接干涉握手过程，但是可以记录下握手的全部数据，之后进行暴力破解以获得预主密钥，并解密所有的信息。一个强大的攻击者可以在数分钟甚至更少的时间内完成攻击；事实上几乎任何人都可以在几个小时内完成这种破解。

幸运的是，现代客户端（例如较新的浏览器）已经不再支持出口密码套件，但是FREAK攻击之所以危害较大，是因为它并不需要浏览器支持这些套件。在正常的RSA握手过程中，ServerKeyExchange消息是不允许发送的。但是有问题的SSL库会继续处理这个消息进而为密钥交换提供弱RSA密钥。①

为了能够进行FREAK攻击，攻击者必须想方设法让客户端接收到一个ServerKeyExchange消息。如果成功的话，可以将连接的安全强度减低到512位。但是这里有两个障碍：(1) 被注入的消息必须被目标服务器上的强RSA私钥签名；(2) 为了修改TLS握手过程中的消息，攻击者必须找到方法来伪造Finished消息，以使得对消息的修改变得合法，即使这本应该是只有拥有私钥的服务器才能完成的工作。FREAK攻击的过程如图6-3所示。

因为第一个障碍，攻击者只能攻击支持使用出口密码套件进行握手的服务器。攻击者首先直接连接到服务器，并只使用出口密码套件与服务器开始握手。这触发了一次使用出口密码套件的握手过程，并导致服务器同时发送ServerKeyExchange消息给攻击者。之后的技巧就是将ServerKeyExchange消息再发送给受攻击的客户端。虽然TLS协议对签名重放攻击进行了防护，但是这种防御依赖客户端和服务器端在ClientHello和ServerHello消息中的随机数。然而，一个主动攻击者可以等到客户端先发送ClientHello之后，将其中的随机数发送给服务器，这样产生的ServerKeyExchange消息就可以绕过客户端的重放攻击检查。

更大的问题是如何生成正确的Finished消息。你应该能想起来这些消息是对所有握手数据进行散列之后并用对称密钥加密得来的。通信双方通过将消息内容与自己计算的数值进行对比来进行校验。攻击者没法直接修改Finished消息内容。然而，在现在这种情况下，攻击者已经成功地将握手的安全强度减弱到512位。如果她可以使用强大的计算资源，则可以实时暴力破解出客户端发出的预主密钥，并获得对连接的全部控制，当然这也包括伪造出合法的Finished消息。

当今，512位的密钥是不安全的，但是它们也不是那么容易被破解。当然，我们有理由相信某些组织可以实时破解掉这些密钥，但是这并不是所有人都需要担心的首要问题。因为事实情况比这还要糟糕。为了性能的原因，相对于为每次握手生成新的密钥，一些服务器只生成一个弱密钥并重复使用一段时间。有时重复使用的时间很长，长到哪怕一个只拥有普通资源的攻击者都有足够的时间来破解它。这也就是之前的那些研究人员做的事情。他们定位了一个重复使用密钥的服务器，然后花100美元在AWS EC2上购买了一些云计算资源，最后用7个小时破解了这个

① 讽刺的是，OpenSSL将这个行为实现成一个“特性”（可以参考SSL_OP_EPHEMERAL_RSA配置选项的文档）。RSA密钥交换不提供向前保密，因为服务器私钥是用来保护全部的预主密钥。因为私钥会静态存在很长时间，任何拿到这个私钥的人都可以破解之前的全部流量。然而，你每次都生成一个全新的RSA私钥，每个连接就会使用不用的私钥加密（虽然私钥强度较弱）。当时，为了这个目的而使用512位的私钥看起来不像今天这样糟糕，或者当时他们也许没有想到某些实现会重用这些弱密钥。

密钥。[①]使用这个密钥，研究人员可以很容易地对有风险的客户端肆意进行中间人攻击，只要这个密钥依然在服务器上保持不变即可。这真是聪明！

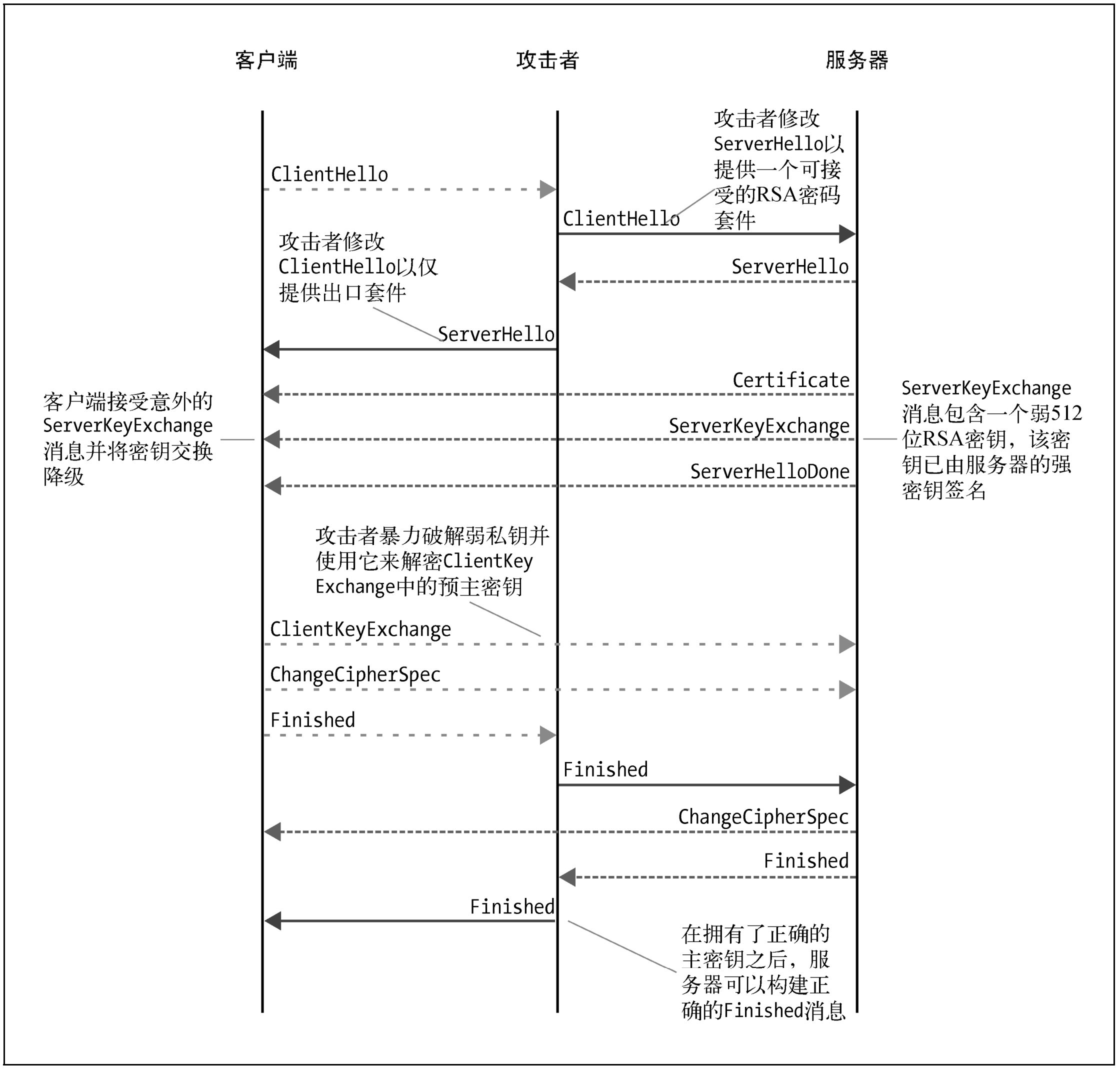


图6-3　FREAK攻击

① Attack of the week: FREAK (or'factoring the NSA for fun and profit'), http://blog.cryptographyengineering.com/2015/03/attack-of-week-freak-or-factoring-nsa.html（Matthew Green，2015年3月3日）。

6.4.3 影响和缓解方法

一开始，人们以为只有OpenSSL会受到FREAK的威胁。虽然在客户端大部分浏览器都不使用OpenSSL，但是OpenSSL有一个很大的客户端用户是Android系统，这使得数以十亿计的智能手机存在潜在的风险。这不仅仅影响浏览器，也影响平台上的其他应用程序。[①]

之后攻击面扩大了，这是因为人们发现Secure Transport（Apple的SSL/TLS库）和Schannel（Microsoft的SSL/TLS库）也存在同样的问题。[②]其他较少使用的库和平台也同样发现了问题。主流浏览器中，唯一不受影响的只有Firefox。

Secure Transport[③]和Schannel[④]在发现问题后就得到了快速的修复，但是即使如此，由于部署的多样性，很难准确地讲哪些使用这些库的产品是不再受到威胁的。例如，在撰写这本书的时候，对于Secure Transport的修复只在OSX v10.8.x和iOS 8.2之后的版本上提供，老的版本依然存在这个漏洞[⑤]。最好的方法就是对你的设备进行一次测试，例如使用SSL Labs提供的工具[⑥]。

> **注意**
>
> FREAK攻击很好地提醒了人们，移除老旧和无用的功能是非常明智的选择。这样可以减小攻击面，也就降低了风险。

对于服务器运维人员的建议是移除对出口密码套件的支持，因为这是完成攻击的必要条件。任何服务器都不应该优先使用这类低强度的套件，但是管理员在移除过时的协议特性时通常会显得比较保守。根据专门对FREAK进行跟踪的网站的数据，互联网上将近四分之一的服务器还在使用过时的密码套件，并可能存在潜在威胁[⑦]。

并不是所有这些服务器都面临同样级别的危险。据估计，大概三分之二的服务器使用的是重复弱密钥，其他的服务器则会为每次握手重新生成新的密钥。[⑧]

---

① HTTPS-crippling FREAK exploit affects thousands of Android and iOS apps，http://arstechnica.com/security/2015/03/https-crippling-freak-exploit-hits-thousands-of-android-and-ios-apps/（Ars Technica，2015年3月17日）。

② Stop the presses: HTTPS-crippling“FREAK”bug affects Windows after all，http://arstechnica.com/security/2015/03/stop-the-presses-https-crippling-freak-bug-affects-windows-after-all/（Ars Technica，2015年3月6日）。

③ About Security Update 2015-002，https://support.apple.com/en-us/HT204413（Apple，2015年3月9日）。

④ Security Advisory 3046015: Vulnerability in Schannel Could Allow Security Feature Bypass，https://technet.microsoft.com/en-us/library/security/3046015.aspx（Microsoft，2015年3月5日）。

⑤ About the security content of iOS 8.2，https://support.apple.com/en-us/HT204423（Apple，2015年3月9日）。

⑥ SSL Labs Client Test，https://www.ssllabs.com/ssltest/（Qualys，检索于2015年3月21日）。

⑦ Tracking the FREAK Attack，https://freakattack.com/（University of Michigan，检索于2015年3月21日）。

⑧ GitHub Gist: Temporary RSA 512 Bit Keylife for FREAK attack，https://gist.github.com/ValdikSS/f4ba45198fe69c349e9c（ValdikSS，检索于2015年3月21日）。

针对SSL和TLS的状态机攻击

FREAK是miTLS发现的若干问题中的一个。miTLS是Microsoft Research-Inria Joint Centre[①]（Microsoft-法国国立计算机及自动化研究院联合研究中心）发起并运行的项目。miTLS是对TLS协议的一个验证参考实现。[②]在开发miTLS的时候，作者们研究并改进TLS协议的安全性。FREAK攻击被包括在他们的关于攻击TLS状态机的工作之中，这项工作已经发现了会引起一个或几个实际攻击的大量的实现缺陷[③]。你可以在他们的网站上找到更多的相关信息：www.smacktls.com。

6.5　Logjam

在2015年1月出现FREAK攻击之后，那些对SSL和TLS协议有着深入理解的人们便知道，出现针对临时Diffie-Hellman（ephemeral Diffie-Hellman，DHE）密钥交换算法的类似攻击只是时间问题。FREAK攻击针对的是RSA密钥交换算法，但是出口密码套件对于DHE也要有一个类似的降级机制（只有512位的安全强度）。如果你能够攻击一个弱的密钥交换机制，那么是否确定也能攻击其他的弱密钥交换机制？然而，当一个叫作Logjam的新漏洞[④]在2015年5月被一个14人的团队发现之后，我们不仅确定了上述问题，而且还有了其他更多的收获。这个攻击的名称是一个双关语，它参考了离散对数问题（discrete logarithm problem，DLP）这个名称，这是构成DH密钥交换机制的一个单向函数。

6.5.1　针对不安全 DHE 密钥交换的主动攻击

最早被发现的攻击，其过程是一个主动的网络攻击者（我们叫她Mallory）首先拦截受害者的TLS连接，然后迫使其使用较弱的DHE密钥交换算法，这种安全强度较弱的算法可以被实时破解。这种攻击复制了FREAK攻击的原理。因为它们的相似性，我不准备花费很多时间来讨论Logjam攻击的细节；相反，我假设你已经对FREAK攻击很了解，因此下面只列出两者的差异。

- 服务器愿意使用不安全的DH参数

  在DHE密钥交换算法中，服务器生成临时密钥并且使用私钥对这些数据进行签名以便进行身份验证。Logjam攻击要想成功，服务器必须愿意使用弱的临时密钥（例如512位）。这种情况在服务器使用出口DHE密码套件的情况下很常见，因为在这种情况下密钥的长

① 这是Microsoft研究院和Inria（一个致力于计算科学研究的公共研究组织）的合作。你可以在网站www.msr-inria.fr上找到更多的信息。

② miTLS，http://www.mitls.org/wsgi/home（Microsoft Research-Inria Joint Centre，检索于2015年3月21日）。

③ A Messy State of the Union: Taming the Composite State Machines of TLS，https://www.smacktls.com/smack.pdf（Beurdouche等，2015年3月）。

④ The Logjam Attack，https://weakdh.org/（Adrian等，发表于2015年5月20日）。

度会被限制在512位。如果条件合适，Mallory可以迫使服务器降级使用弱的DHE密钥交换方法并对其进行破解，这与FREAK的方法是完全一样的。

在某些情况下，如果服务器本身使用了不安全的DHE参数，不安全的服务器也可能被动地遭受攻击。然而，这很不常见，因为大多数客户端不支持出口密码套件，而这种套件展现出了最大的攻击面。此外，大部分服务器更倾向于使用速度更快的ECDHE和RSA密钥交换算法。这意味着使用DHE的可能性会比较小。

- 服务器缓存临时DH密钥

 为了使一个主动攻击奏效，Mallory必须拦截原始的TLS握手，并从中迫使其使用不安全的加密，然后生成正确的`Finished`消息以避免被发现。这意味着她必须在一分钟内破解512位的密钥，这是大多数现代浏览器对TLS握手失败的超时等待时间。在这些限制下，实施攻击是非常困难的，即使对于一个装备精良的攻击者来说也是如此。

 然而，某些服务器会很不明智地缓存它们的“临时”密钥。对于存在这种行为的服务器，Mallory可以实现破解临时密钥并在后续使用这个结果来进行实时攻击，攻击的有效时间范围等同于服务器缓存密钥的时间。当临时密钥发生变化的时候，如果Mallory需要继续进行攻击，就可以再次破解密钥。

 援引自论文，大概有15%服务器会缓存密钥，但是只有0.1%左右的服务器使用可以被实时破解的512位DH参数。值得注意的是，Microsoft IIS将会缓存2个小时的密钥。OpenSSL中有一个开关来控制这个行为，但是最流行的Web服务器（Apache和Nginx）都不使用这个行为。

- 客户端愿意接受弱DH参数

 Logjam只在客户端会使用弱DH参数的时候才能奏效。不幸的是，在发现攻击时，所有主流浏览器都接受使用512位的不安全密钥交换。攻击甚至对于那些不使用出口DHE套件的客户端起作用，这是因为DH套件（无论是否是出口）没有指定密钥交换的强度。在实际中，密钥交换的强度是由服务器来决定的。

虽然可以说是浏览器（或者其他客户端）接受不安全的DH参数才使得Logjam攻击成为可能，但是这个攻击也揭示出了TLS协议中的一个缺陷。可以回忆一下，FREAK攻击利用了客户端的TLS协议实现缺陷，该缺陷使得协议上不允许出现`ServerKeyExchange`的环节中可以出现这条消息。Logjam不需要利用实现上的缺陷，因为它利用的是协议上的弱点：服务器对于DH参数的签名可以被重放。对于签名的设计，TLS确实有重放保护机制，但是这个保护机制只依赖唯一的客户端和服务器端随机数。问题是，一个主动攻击者可以通过观察客户端提供的值并将它使用在不同的握手中来同步这些随机数。关键的是，这个缺陷可以导致攻击者强迫握手协商出原本不会使用的密码套件。

6.5.2 针对不安全 DHE 密钥交换的预先计算攻击

6.5.1节所描述的针对不安全DHE密钥交换算法的主动攻击很巧妙，但是受限于服务器需要允

许512位的不安全DH参数和对临时密钥的缓存。对于更高安全强度的攻击是不可行的。在攻击被发现的时候，也有服务器在使用768位的DH参数，但是更多的服务器在使用1024位参数。直接攻击这种强度是没有办法的，即使对于政府机构。然而，这就是*预先计算攻击*（precomputation attack）诞生的原因。

为了理解预先计算攻击，你需要知道DH密钥交换算法是基于双方协商出的域参数来执行各种操作的。这个域参数包含两个值：一个是素数p，另外一个是生成元g。实际来讲，这些参数是用来扰乱密钥交换过程的，并会使某些数学计算对于攻击者来说很难。Mallory知道这些参数，因为它们是公开的（例如，在TLS 1.3之前，这些参数由服务器通过网络发送），但是这并不能帮助完成攻击。为了能破解密钥交换，Mallory必须要知道握手过程中生成的两个临时密钥之一。在本节的剩余部分，我将假设她选择的是服务器端密钥y。

当前已知的对DH密钥交换的最佳方法是使用*数域筛子*（number field sieve，NFS）。对Logjam攻击的研究所获得的突破性进展是，NFS可以分成两个主要步骤。第一个步骤是最费力的，但关键的是，这个步骤中的计算只依赖参数中的素数。这意味着，一旦第一步产生了结果，就可以立即用来破解任何使用相同素数的密钥交换过程中产生的临时密钥。为了进行说明，论文的作者们进行了一项试验，在试验中，针对512位素数的预先计算攻击耗费了一周的时间。在这之后，论文的作者们可以在60秒内对任何单独的密钥交换（使用相同的素数）进行攻击。

当考虑到预先计算的时候，针对512位的DH参数的主动攻击是几乎人人可以完成的。但是，768位和1024位的强度又会有多么安全呢？

6.5.3　针对弱 DH 密钥交换的状态–水平威胁

基于他们自己破解512位DH参数的经验，并结合实验以及来自其他来源的可靠信息，研究者们估计出了破解掉768位和1024位参数所需的大概时间。你可以从下面的表6-2中看到结论。很明显，随着安全强度的升高，攻击者的开销会显著增大。

表6-2　针对弱DH参数进行攻击的估计开销（来源：Adrian 等）

	预先计算（核心年）	单次攻击（核心时间）
DH-512	10	10分钟
DH-768	36 500	2天
DH-1024	45 000 000	30天

然而，考虑到这些计算都是可以高度并行的以及攻击可以进一步优化，研究者总结出破解768位的参数对于学院研究者来说是触手可及的。对于1024位的参数，研究者估计可以完成任务的特殊硬件的开销应该在数亿美元这个量级，然后攻击者需要花费一年的时间来完成破解。

此刻，你可能会认为花费数亿美元只为了破解一个1024位的参数是很昂贵的。你有可能是对的。不幸的是，攻击者的工作因为很多应用程序重用所谓的*标准组*（standard group）而变得简单。这也就是说很多服务器都在使用同一个素数。正是因为这一点，一次预先计算的结果可以用在上千甚至上万的服务器上。

> **注意**
> 在Logjam攻击被发现之后，使用标准组看起来不是很安全，但是依然有很好的理由来继续使用它们。最近关于安全协议的经验显示，协议的设计者应该尽可能多地完成自己的工作。当实现者不需要考虑过多的时候，他们犯错误的概率也会相应减小。例如，即使在Logjam的研究报告中也有两个小节是专注于如何利用生成的DH参数进行攻击的。如果使用的是标准参数，这些问题就不会发生。Logjam发现的真正问题是当下使用的标准参数的强度不够。

6.5.4 影响

Logjam的影响基于不同的攻击形态而区别很大。在最坏的情况下，服务器默认使用DHE密钥交换算法并使用512位的参数。这样被动攻击者会轻易地破解他们的流量并记录下会话内容。下一种场景是服务器使用了不安全的参数，但是默认会优先使用ECDHE或者RSA密钥交换算法。这些服务器可能被主动攻击者攻击，但是首先它们要成为足够重要的攻击目标。如果这些服务器使用的是通用DH参数，归功于预先计算攻击，进行主动攻击的开销会小得多。

根据研究者的结论，攻破了1024位参数的攻击者可以对大约6.56%的HTTPS服务器进行被动攻击，而攻破了10个通用组的攻击者可以对大约10%的服务器进行攻击。一个主动的攻击者可能分别攻破12.8%和23.8%的服务器。

> **注意**
> 作为一个主动攻击者，要想攻击成功，Mallory必须在最多1分钟内攻破密钥交换过程。然而，现代浏览器支持一个叫作False Start的机制，这种机制为了性能而减弱了安全性。这种浏览器在校验TLS握手完整性之前就发送了HTTP请求。如果针对这类浏览器进行攻击，Mallory也许没必要实时攻破密钥交换。因为HTTP请求通常含有敏感信息（例如，Cookie和密码），这对后续破解密钥交换可能是有用的。有关False Start的更多信息，请参考9.2.1节中的False Start部分。

针对TLS协议之外的被动攻击更加有趣。像SSH和IPsec这类协议处理大量的流量，它们也倾向于使用1024位的DH参数，通常是标准组。对于SSH，举例来说，作者们估计攻破了一个标准组（1024位Oakley Group 2）的攻击者可以被动地攻击25.7%（360万）的服务器。对于IPsec，情况更加糟糕，同样的攻击者可以攻击超过60%的服务器。

NSA在破解1024位的Diffie-Hellman密钥交换吗？

Logjam的作者们相信NSA有能力破解至少一小部分1024位DH参数组，但是NSA是否在做这些事情呢？鉴于NSA的年度预算超过100亿美元，他们是有这种财力的。Logjam攻击和早期

的发现吻合：NSA看起来可以解密大量的VPN流量。这个信息来源于由爱德华·斯诺登泄露的机密文件，并由德国《明镜周刊》在2014年12月发表。[①]虽然文档没说明加密数据是如何破解的，但是却展示了系统架构和输入需求。其中就包括对1024位Diffie-Hellman密钥交换算法的有效破解。

6.5.5　缓解方法

尽管存在潜在的损失，Logjam也是很容易缓解的。第一步，禁用出口套件这个老古董，这对于解决紧急问题（容易被攻击）来说通常足够了。第二步，如果在你的配置中存在某些DHE套件，确保它们使用的是2048位的参数。不要使用低于1024位的参数。

取决于你的目标用户，将DH参数的强度增强到2048位不是一蹴而就的。主要的问题是Java 6客户端不支持高于1024位强度的参数。[②]

如果你纠结于支持Java 6还是使用更高强度的算法，有两个选择。第一，依然使用1024位的参数，但是确保没有使用标准组：为每个服务器生成一个独立的参数。

第二，完全禁用DHE算法。这是因为DHE并不是一个很流行的密钥交换算法。很多人不喜欢DHE是因为该算法比其他算法要慢，例如RSA和ECDHE（如果你对性能很好奇，可以参考第9章）。但是你不应该使用RSA算法，因为它不提供前向保密，因此ECDHE就变成了绝对的赢家，因为它兼顾了性能和安全性。因为几乎所有现代浏览器都支持ECDHE，使得禁用DHE并不会影响你的安全。唯一可能的影响就是一小部分的流量可能会丢失掉前向保密。

被发现支持不安全DH参数的那些浏览器厂商已经作出保证要将最小安全强度增强到一个安全的值。IE浏览器已经将强度增加到1024位；Chrome和Firefox也将会跟随这一做法。Safari则增强到768位。OpenSSL将默认强度提高到768位，并在不远的将来会提高到1024位。

6.6　协议降级攻击

协议降级攻击是指攻击者作为中间人企图修改TLS握手过程中的连接参数。具体思路是，攻击者希望迫使握手使用一个低等级的协议或者使用低强度的密码套件。在SSL 2中，这类攻击是很容易发生的，因为该协议不提供握手的完整性校验。后续的SSL协议则提供握手完整性校验以及其他的附加手段来检测类似的攻击。

然而，协议设计者没有想到的是由协议演进而带来的协议互操作性问题。浏览器会尽最大可能与所有服务器进行通信。不幸的是，在TLS协议上，这种尝试会导致安全风险，因为浏览器为了兼容服务器，会降低它们的安全能力，也就是为了互操作性而降低安全性。

---

① Prying Eyes: Inside the NSA’s War on Internet Security，http://www.spiegel.de/international/germany/inside-the-nsa-s-war-on-internet-security-a-1010361.html（《明镜周刊》，2014年12月）。

② Java 7客户端也不支持。然而它们默认支持ECDHE密钥交换算法，这意味着你可以将服务器配置成使用ECDHE并永远不去尝试DHE算法。

6.6.1 SSL 3 中的回退保护

在SSL 2中，没有机制来保证握手的完整性，因此该协议存在降级攻击的风险。结果，作为中间人的攻击者总是可以迫使一次握手使用安全强度最低的协议参数。握手完整性校验在SSL 3中引入，这也是SSL 3协议的一个主要完善之处。

但是为了在SSL 3中提供握手完整性校验（也包括其他的一些改进），SSL 3必须修改初始握手请求的格式（`ClientHello`）。此外，可以理解新协议的服务器会自动升级到新的格式与客户端兼容，如果客户端也同样支持的话。但是依然遗留了以下几个问题。

(1) 虽然SSL 3协议提供了完整性检查，但是实际上你无法使用这个功能，因为大部分服务器只能处理SSL 2。

(2) 即使是支持SSL 3的服务器，如果有主动的中间人攻击者，他依然可以截获连接并对客户端伪装成只支持SSL 2的服务器。

(3) 如果你随后使用了SSL 2握手，那么就没有了协议完整性校验，中间人攻击也会成功。

为了应对这些漏洞，SSL 3引入了协议回退保护机制[①]，该机制可以让支持SSL 3的客户端和服务器能检测出它们正遭受攻击。当一个SSL 3的客户端因为兼容性问题回退到SSL 2时，他将RSA密钥的PKCS#1块按照特殊方式重新编排。[②]在SSL 2中，块结尾必须包含至少8字节的随机数据；SSL 3客户端则用`0x03`填充这8字节。这样，如果一个SSL 3的客户端被强迫降级到SSL 2，则SSL 3服务器会注意到这个特殊的格式，从而发现攻击并结束握手。仅仅支持SSL 2的服务器不会读取这些填充数据，因此握手可以继续进行。

还有一个漏洞可以破坏回退保护机制。在SSL 2中，主密钥的长度反映了协商出的密码套件的长度；在最坏的情况下，只有40位。此外，是客户端来选择服务器支持的密码套件，生成主密钥，将其用公钥加密后发送到服务器。服务器用RSA私钥解密消息，获取主密钥，然后向客户端证明身份。

对于中间人攻击者来说，暴力破解RSA密钥工作量较大，但是能攻击较弱的主密钥。他可以伪装成服务器并仅仅提供40位的套件，然后暴力破解主密钥，最后完成握手。在当今的计算能力下，这种破解是很简单的。这种攻击的影响在今天几乎可以忽略，因为只有极少的客户端还在支持SSL 2。结论是SSL 2依旧无法提供大于40位的安全性。攻击者如果可以实时进行暴力破解，就可以攻击所有的SSL 2连接。

6.6.2 互操作性问题

随着SSL 3的公布，互操作性问题也随之而来。在本节中我将列举一些最常见的问题。

① RFC 6101: The SSL Protocol Version 3.0, Section E.2，http://tools.ietf.org/html/rfc6101#appendix-E.2（Freier等，2011年8月）。

② 在SSL 2中，RSA是唯一的身份验证和密钥交换机制。因此作为对这种密钥交换的修改而实现的回滚保护机制足够解决这个问题了。

1. 版本不相容问题

遇到的第一个问题是版本不相容。SSL 2不需要考虑协议进化而且也没有规定如何处理未知的协议版本。以下书摘[1]描述了这种情况。

> 不幸的是，SSLv2规范并不清楚服务器应该如何处理版本号比它们能支持的版本号更高的CLIENT-HELLO消息。由于Netscape的SSLREF参考实现简单地拒绝掉所有更高版本号的握手请求的事实，使得这个问题变得更加糟糕。因此，不能保证所有的SSLv2服务器都可以正确地处理向后兼容的握手请求，虽然大部分的主流服务器都是支持的。

SSl 3没有在这方面进行大幅改进，只是在协议规范中用一句话提到了客户端版本处理方法：

> `server_version`：这个字段包含client hello中客户端建议的版本和服务器支持的最高版本中的较小者。

从TLS 1.0开始，有了更多关于如何处理向后兼容性的描述，但是只有TLS 1.2提供了清晰的指导。

> 一个TLS 1.2客户端需要与老旧服务器协商的时候，需要发送正常的TLS 1.2 `ClientHello`消息，在`ClientHello.client_version`中包含`{3,3}`（TLS 1.2）。如果服务器不支持这个版本，则它需要在响应的`ServerHello`中包含一个更旧的版本号。如果客户端同意使用这个版本，则协商会继续按照商定的协议版本酌情进行。

由于这些规范的歧义性，很多服务器对于不支持的版本采用了拒绝握手的处理。当浏览器开始支持TLS 1.2的时候，这成为了一种严重的互操作性问题。因为这个原因，IE浏览器，作为第一个支持TLS 1.2的浏览器，在发布的时候默认将TLS 1.1和TLS 1.2禁用。

重新协商指示扩展（Renegotiation Indication Extension）规范（2010年公布，比TLS 1.2晚两年），尝试来解决这个问题。寄希望于开发者在实现新的重新协商机制的时候，也会关注版本和扩展的不相容问题。在规范的3.6节中写道：

> 实现本规范的TLS服务器必须忽略任何由客户端提供的未知扩展，并且必须处理比它们支持的最高版本号更高的版本号并以最高公共版本号进行协商。这两个需求重申之前存在于RFC 5246中的需求并且仅为了有利于向前兼容而在此处声明。

还有一个类似的不相容问题和记录层的TLS版本有关。除了在ClientHello中存在版本号之外，客户端同时也决定在TLS记录中使用哪个版本号，这个信息是在发送给服务器的第一条消息里携带的，在这个环节上，客户端还不知道服务器支持哪些版本。对于记录层版本号的混淆更加的复杂，并且很多服务器显示出了巨大的不相容性。在实际中，某些库永远指定TLS 1.0，反之其他库则指定它们支持的最高版本，就像在ClientHello中的那样（例如，TLS 1.2）。

这种类型的不相容给TLS 1.3造成了很严重的问题。根据我简单的测试，如果将这个新版本号用于记录层，那么大量的服务器（很容易超过10%）将会拒绝进行握手。使用TLS 2.0作为下一

[1] *SSL and TLS: Designing and Building Secure Systems*, 137页（Eric Rescorla，Addison-Wesley，2000年10月）。

个协议版本是完全不可能的，因为不相容性会影响超过70%的服务器。为了将这个问题最小化，TLS 1.3的协议规范废弃了记录层协议版本并将其固定为TLS 1.0。

2. 扩展不相容问题

SSL/TLS协议的早期版本（SSL 3和TLS 1.0）没有在不引入新版本号的情况下增加新功能点的明确机制。唯一类似于向前兼容性的机制是一个关于可以在ClientHello消息结尾增加额外数据的规定。具体的实现会忽略掉不能理解的额外数据。这个模糊的扩展机制后来被TLS扩展所替换掉[①]，TLS扩展为ClientHello和ServerHello消息增加了通用的扩展机制。在TLS 1.2中，TLS扩展被合并到主协议规范中。

由于早期协议规范的模糊性，毫无意外将会有大量的SSL 3和TLS 1.0服务器拒绝与带有扩展数据的客户端进行握手。

3. 其他互操作性问题

还存在其他互操作性问题，这些问题大多是由于模糊的规范和粗心的编程共同导致的。

- 长握手不相容性

  协议本身没有限制ClientHello消息的大小，但是早年协议中只有少量的密码套件可供客户端选择，因此ClientHello消息较短。但是从OpenSSL 1.0.1开始，情况则发生了变化，主要是支持了更多的密码套件。那么把这些大量的密码套件与其他扩展信息一起描述出来时（例如主机名和支持的椭圆曲线类型），ClientHello消息的大小会增长得很大。然后就发生了一个问题，F5的BIG-IP负载均衡器无法处理长度在255~512字节之间的握手消息。因为BIG-IP的广泛使用（特别是那些大网站），使得此问题影响了TLS 1.2的推广。

- 任意扩展不相容

  有时候，能识别TLS扩展的服务器也会不明原因地无法协商出带有不明扩展的链接。这种情况在服务器名称指示（server name indication，SNI）和状态请求（status request；OCSP stapling）扩展上常有发生。

- 无法处理碎片

  历史上，有过许多与消息碎片相关的问题。SSL和TLS协议允许所有上层协议的数据进行碎片整理并且经过多个记录协议消息发送。多数产品可以正确地处理应用层的碎片数据，但是无法处理其他类型的碎片，原因是这类数据碎片的场景在实际中从未发生。同样，一些产品无法处理0长度的TLS记录，这类记录最开始是为了应对在TLS 1.0以及更早的协议中的可预测IV问题。为了应对同样问题的另一个方法是采用$1/n-1$分割的方式发送记录消息（发送两个记录消息，第一个消息包含1字节）。这种方法也会导致问题，因为一些产品无法处理一个HTTP请求横跨两条TLS消息的情形。

① RFC 3546: TLS Extensions，http://www.ietf.org/rfc/rfc3546.txt（Blake-Wilson等，2003年6月）。

6.6.3 自愿协议降级

当互操作性问题出现之后，浏览器开始支持自愿协议降级（voluntary protocol downgrade）。具体来说就是首先使用其支持的最高TLS版本尝试连接，并启用所有支持的选项和扩展。如果首次尝试连接失败，则减少选项并降低协议版本；浏览器会持续进行这种尝试直到连接建立。例如，当TLS 1.0协议为最高版本的时候，自愿协议降级在最坏的情况下会尝试2次，而对于TLS 1.2，则会尝试3 ~ 4次。

> **注意**
> 互操作性问题不仅仅是导致TLS握手失败的原因。有充足的证据表明代理服务器、防火墙以及杀毒软件经常基于协议版本号和其他的握手属性来拦截和过滤连接。

为了了解这种行为，我调查了多种版本的主流桌面浏览器。我使用了一个修改过的TCP代理，该代理只允许SSL 3.0的连接通过。其他版本的连接都会被阻断并返回`handshake_failure` TLS警报。调查结果如表6-3所示。

表6-3　2014年7月主流浏览器的自愿协议降级行为

浏　览　器	第一次尝试	第二次尝试	第三次尝试	第四次尝试
Chrome 33	TLS 1.2	TLS 1.1	TLS 1.0	SSL 3
Firefox 27	TLS 1.2	TLS 1.1	TLS 1.0	SSL 3
IE6	SSL 3	SSL 2		
IE7（Vista）	TLS 1.0	SSL 3		
IE8（XP）	TLS 1.0（无扩展）	SSL 3		
IE8~10（Win 7）	TLS 1.0	SSL 3		
IE11	TLS 1.2	TLS 1. 0	SSL 3	
Safari 7	TLS 1.2	TLS 1. 0	SSL 3	

我的调查结果显示，在2014年7月，你可以将所有主流浏览器降级到SSL 3。[①]对于IE6，你甚至可以降级到SSL 2。基于SSL 2可以收到暴力破解主密钥的问题的威胁，你也只能期望IE6提供最大40位的安全强度了。

对于SSL 3，因为POODLE攻击的原因，该版本已经在2014年10月被明确判定为不安全的。一次成功的攻击可以利用协议的弱点获取到少量的加密数据（例如HTTP协议的Cookie）。即使忽略掉POODLE攻击问题，SSL 3依然在以下方面与最新的TLS 1.2存在巨大差距。

- 不支持GCM、SHA256和SHA384套件。
- 不支持椭圆曲线加密方法。当面临前向保密问题时，只有很少的站点支持不带椭圆曲线的临时Diffie-Hellman密钥交换。没有椭圆曲线，这些网站无法提供前向保密。

① 即使是Opera，其在之前已经实现了协议降级保护机制，但是开发团队在版本15的时候丢弃了自己的引擎而转而使用Chrome的Blink，之后该特性也随之丢失了。

- SSL 3容易遭受到BEAST攻击，不过现代浏览器都实现了防御的手段。然而，有些网站倾向于在TLS 1.0和之前的协议中使用RC4。对于这些网站，攻击者可以迫使连接降级到使用RC4。
- Microsoft的SSL 3不支持AES算法，这意味着使用SSL 3的IE浏览器只能使用RC4和3DES算法。

在上面各项中，我认为最大的问题是不支持前向保密。一个严重的攻击可以导致某些连接降低到使用RSA密钥交换的方法，然后在得到私钥后获取全部加密信息。

> **注意**
> 考虑到通信失败的确切本质，即使在服务器不存在不相容问题的情况下，依然可能会触发回退机制。举例来说，有报道称Firefox在某些不可靠网络连接的情况下，会回退使用SSL 3，最终导致使用虚拟安全托管的站点断开连接（因为虚拟安全托管依赖TLS扩展，但是SSL 3并不支持）。①

基于我的测试，现代浏览器做了很多安全改进。例如，在POODLE攻击被发现后，浏览器首先停止了向SSL 3的降级，然后完全禁用这个版本的协议。行为在不断的变化，并且变化是向着完全不会回退的方向发展。你可以将这里讨论的降级行为作为一种对补救中老旧问题的描述。

6.6.4 TLS 1.0 和之后协议的回退保护

因为SSL 3以及更新的协议提供了握手完整性检查，针对仅支持SSL 3和更高版本的回退攻击无法奏效。②

在SSL 2上起作用的暴力破解主密钥的方法也不再有用，因为主密钥在新版本中提高到了384位。

TLS 1.0（以及所有后续协议）同时也继承了SSL 3的传统并包含了针对RSA密钥交换的回退保护，这是通过让客户端发送一个额外的版本号并由服务器私钥进行保护。以下是引自TLS 1.2规范7.4.7.1节的一段话：

> PreMasterSecret中的版本号是由客户端在ClientHello.client_version中提供的版本号，而不是连接协商出的协议版本号。此特性如此设计是为了防止回退攻击。

这个保护机制只有在RSA作为身份验证和密钥交换的时候才能被使用，它无法用于其他密钥交换算法（即使RSA用于身份验证时也不行）。

实际上，TLS/SSL协议的开发人员经常无法在正确的位置上使用正确的协议版本。Opera的

① Bug #450280: PSM sometimes falls back from TLS to SSL3 when holding F5 (which causes SNI to be disabled)，https://bugzilla.mozilla.org/show_bug.cgi?id=450280（Bugzilla@Mozilla，报告于2008年8月12日）。

② 保护是由于Finished消息提供的，该消息用来在握手结束的时候进行完整性校验。在SSL 3中，这个消息的长度为388位。奇怪的是，TLS 1.0将此消息的大小削减为96位。在TLS 1.2中，Finished消息仍然默认使用96位的长度，但是协议规范允许密码套件增加长度。尽管如此，所有的密码套件依然在使用96位的长度。

SSL/TLS库的维护者Yngve Petterson曾经在这个问题上表示：[①]

> 第二，基于RSA的密钥交换算法使用了这样一种方法，客户端同时将它发送给服务器的协议版本号再发送一次，服务器将此版本号与之前收到的版本号进行对比检查。这可以保护协议版本选择，即使在散列函数的安全性被破坏的情况下同样有效。不幸的是，很多客户端和服务器没有正确地按照规范实施，使得这个保护机制没有任何效果。

在TLS 1.2的规范中也有同样的描述：

> 不幸的是，某些旧的实现使用了协商出的版本号，因此检查版本号可能会导致无法与这些错误实现的客户端建立连接。

紧接着该规范给出只针对新的客户端启用回退保护的建议：

> 如果`ClientHello.client_version`是TLS 1.1或者更高，服务器实现必须按照如下面的说明中所述的方法来检查版本号。

不过，虽然存在两种防御方法，但是协议回退攻击依然可能发生，这是因为之前我们讨论过的客户端自愿协议降级行为。

6.6.5　攻击自愿协议降级

协议内置的针对回退攻击的防御对于攻击者干扰一个新连接进行攻击是有效的。但是当考虑自愿协议降级的情况后，回退攻击依然会发生。这是因为中间人攻击者根本不需要修改任何握手数据。相反，他只需要拦截掉客户端发出的高于SSL 3的握手请求，迫使客户端降低协议版本。为了防御这种类型的攻击，需要使用新的办法。

6.6.6　现代回退防御

自愿协议降级行为是TLS安全性的一个漏洞。虽然大家都努力升级到TLS 1.2，但是主动攻击者仍然可以将通信降级到TLS 1.0，甚至在某些情况下降级到SSL 3。这个话题在TLS工作组的邮件列表上讨论了很多次，但是到目前还没有达成任何共识。我收集了一些与讨论有关的链接，意在了解人们关于解决这个问题的思路以及观察工作组工作的复杂性。

这个话题最早是在2011年提出的[②]，当时Eric Rescorla提议使用特殊的信号密码套件值（signaling cipher suite value，SCSV）来让客户端用它们支持的最高版本协议进行通信，即使在出现协议降级的情况下也是如此。服务器在发现协议版本号不符合的时候必须关闭连接。这基于的假设是支持这种防御功能的服务器不能存在任何的不相容问题。选择SCSV方法的原因是它同时

---

① Standards work update，https://web.archive.org/web/20140301045454/http://my.opera.com/yngve/blog/2012/11/02/standards-work-update（Yngve Nysæter Pettersen，2011年12月2日）。

② One approach to rollback protection，https://www.ietf.org/mail-archive/web/tls/current/msg08099.html（Eric Rescorla，2011年9月26日）。

成功地支持了在SSL 3基础上的重新协商。[①]

2012年，Adam Langley提出了一种同样是基于信号套件并且将检测方法留在服务器端的防御方法。[②]

在之后的讨论中，Yngve Peterson提交了一种替代方法[③]，倾向于在客户端进行检查[④]（这将使得实现起来更加容易；与要使用很长时间来升级大量服务器相比，只需要升级有限种类的客户端就能节省很多时间）。他的提案基于RFC 5746（重新协商指示扩展），该规范明确禁止了服务器对未来协议版本的不相容性。Yngve的评估指出，只有0.14%的实现了RFC 5746的服务器显示出了协议不相容性。他随后在Opera 10.50中加入了这个防御回退攻击的方法。[⑤]

另外一次讨论在2013年4月展开[⑥]。最终，Bodo Moeller在2013年9月提交了一份草稿[⑦]。该草稿后来得到完善[⑧]，并且最终成为了RFC7507[⑨]。Bodo的提案主要是使用了一个单独的信号套件来指示自愿降级行为。需要理解该信号并支持更高协议版本（与客户端将尝试协商的协议版本相比）的服务器来中止协商。Chrome 33是实现此特性的第一个浏览器[⑩]。

6

我们该如何解释工作组对于Yngve的提案缺乏兴趣呢？可能是因为尽管十分罕见，但依然有实现了安全重新协商的服务器存在协议不相容的问题。我认为浏览器厂商们不希望事态发展到最终必然对他们产生强烈影响的程度。另外一方面，SCSV方案会在服务器端实施并只针对真正的攻击起作用。

SCSV方案的问题在于推广需要花费很多年。少数关心安全的网站可能会尽快地部署该方案，但是其他的网站则会因为开销太大而无法部署。Google在2014年2月开始使用SCSV，同时在Chrome浏览器和服务器端实现。OpenSSL 1.0.1j版本是2014年10月发行的，包括了服务器端对这

① 对于现代的协议版本，客户端可以用TLS扩展来声明它们的能力。但是因为SSL 3不支持扩展，这就需要其他的机制。方法就是使用信号套件，这种套件不能用来协商，但是可以实现从客户端到服务器的少量信息传送。

② Cipher suite values to indicate TLS capability，http://www.ietf.org/mail-archive/web/tls/current/msg08861.html（Adam Langley，2012年6月5日）。

③ Fwd: New Version Notification for draft-pettersen-tls-version-rollback-removal-00.txt，http://www.ietf.org/mail-archive/web/tls/current/msg08889.html（Yngve Pettersen，2012年7月3日）。

④ Managing and removing automatic version rollback in TLS Clients，https://datatracker.ietf.org/doc/draft-pettersen-tls-version-rollback-removal/（Yngve Pettersen，2014年2月）。

⑤ 从版本15开始，Opera切换到了Blink浏览器引擎（Google的WebKit分支），废弃了他自己的引擎以及SSL/TLS栈。这也意味着同时废弃了Yngve所提议的那个回滚实现。

⑥ SCSVs and SSLv3 fallback，http://www.ietf.org/mail-archive/web/tls/current/msg09450.html（Trevor Perrin，2013年4月4日）。

⑦ TLS Fallback SCSV for Preventing Protocol Downgrade Attacks，https://tools.ietf.org/html/draft-bmoeller-tls-downgrade-scsv-02（Bodo Moeller和Adam Langley，2014年6月）。

⑧ An SCSV to stop TLS fallback，http://www.ietf.org/mail-archive/web/tls/current/msg10676.html（Adam Langley，2013年11月25日）。

⑨ RFC 7507: TLS Fallback SCSV for Preventing Protocol Downgrade Attacks，https://datatracker.ietf.org/doc/rfc7507/（Moeller等，2015年4月）。

⑩ TLS Symmetric Crypto，https://www.imperialviolet.org/2014/02/27/tlssymmetriccrypto.html（Adam Langley，2014年2月27日）。

个新标准的支持。Mozilla计划在Firefox 35中实现SCSV，预计在2015年1月发行。[①]

作为对比，IE浏览器团队声明他们没有计划来实现这个防御机制。[②]

6.7　截断攻击

在截断攻击中，攻击者可以提前结束一条安全连接，以阻断一条或多条消息到达对端。通常来说，一个安全协议是应该能检测到这类攻击的。SSL 2会受到截断攻击的威胁，但是SSL 3通过引入close_notify消息来处理这个问题。之后的协议版本保留了这个保护机制。例如，TLS 1.2协议规范的7.2.1节包含了以下说明：

> 除非有其他更严重的警报消息，否则通信的双方需要在关闭连接的写方向之前，发送close_notify消息。收到此消息的通信另外一方必须用它自己的close_notify警报作为响应，并立即关闭连接且忽略掉所有正在等待的写事件。

这个机制是有效的，因为close_notify消息是经过验证的。如果任何之前消息产生了丢失，则TLS的完整性检查机制可以发现这个问题。

不幸的是，连接关闭危害一直在大范围地扩散。这是因为很多客户端和服务器会突然关闭连接并省略掉了标准中强制性的连接关闭流程。IE浏览器就是这样，此外还有很多其他的软件同样如此。

由于存在很多关于截断攻击的虚假报警，一些运行良好的应用程序开始忽略这个问题，这使得它们开始面临真正的攻击。

实际上，各种标准鼓励了这种行为，因为它们并没有真正对可靠的关闭连接进行严格要求。SSL 3的规范中有如下描述：

> 要求发起关闭连接的一方在关闭连接的读方向之前等待对端的close_notify警报。

也就是说，不需要操心对端是否收到了本端发送的全部数据。由于TLS 1.1放松了会话恢复的规则，事情变得更糟。在TLS 1.1之前，任何连接上的错误都会导致TLS会话信息被丢弃。在实践上，这表示客户端需要在后续的连接上重新执行全握手（CPU密集型操作）流程。但是TLS 1.1对于非正常关闭的连接删除了这个要求。下面是引用自TLS 1.1的7.2.1节中的一段话：

> **注意**
> 对于TLS 1.1，无法正确关闭连接不再要求不恢复会话。这与TLS 1.0不同，目的是为了符合广泛应用的实现实践。

这很可惜，因为这次修改让那些存在错误的浏览器失去了更正问题的动力。结果就是，我们

---

① The POODLE Attack and the End of SSL 3.0，https://blog.mozilla.org/security/2014/10/14/the-poodle-attack-and-the-end-of-ssl-3-0/（Mozilla安全博客，2014年10月14日）。

② Internet Explorer should send TLS_FALLBACK_SCSV，https://connect.microsoft.com/IE/feedback/details/1002874/internet-explorer-should-send-tls-fallback-scsv（IE Feedback Home，提交于2014年10月16日）。

失去了对截断攻击的防御。

6.7.1 截断攻击的历史

针对SSL 3和TLS的截断攻击最早在2007年就被讨论过[1]，当时是Berbecaru和Lioy演示了针对多种浏览器的这类攻击。他们把焦点集中在截断响应上。例如，浏览器会通过TLS连接显示不完整的页面或者图像，却不会发出任何关于当前页面内容不完整的提示。

这个话题在2013年被重新提及[2]，这次比之前讨论得更加详细。具体来讲，Smyth和Pironti成功地展示了几个引人注目的攻击，攻击的范围包括从在公共环境中针对电子投票系统（Helios）的攻击到针对Web邮件系统账号的攻击（Microsoft和Google）。在所有的攻击场景中，最主要的技巧是在用户不知道的情况下阻止用户的账户登出。为了达成这点，他们利用应用程序的漏洞告知用户已经登出，但实际上用户并没有登出。通过利用针对HTTP请求的TLS截断，研究者可以让被攻击用户保持登录状态。在这之后，如果攻击者能够登录到被攻击人的电脑上，那攻击者就可以伪装成被攻击人并利用他的身份。

> **注意**
>
> 截断攻击对HTTP协议有效，这很有意思，因为HTTP消息是被设计成包括长度信息的，但是却没什么效果。这是在互联网领域人们急功近利地走捷径而最终产生问题的又一个例子。

6.7.2 Cookie 截断

2014年，一些更加有效的截断攻击新技术诞生了。[3]研究者将早期针对TLS的攻击方法（例如BEAST攻击）应用到了截断攻击上，具体来说，攻击者可以通过向HTTP请求和响应中注入任意长度的数据来达到控制TLS记录长度的目的。如果攻击者能控制TLS记录的长度，那么就可以控制记录如何进行分割（取决于大小和其他限制）。在结合截断攻击的情况下，攻击者可以分割HTTP的请求和响应头，这会造成一些有意思的结果。

Cookie截断是一种针对HTTP响应头的截断攻击。这种攻击可以让安全Cookie变成明文的、不安全的Cookie。我们来看一些含有安全Cookie的HTTP响应头：

```
HTTP/1.1 302 Moved Temporarily
Date: Fri, 28 Mar 2014 10:49:56 GMT
Server: Apache
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

---

① 关于基于SSL和TLS安全协议的应用程序的可靠性（Diana Berbecaru和Antonio Lioy，*Public Key Infrastructure, Lecture Notes in Computer Science*，第4582卷，第248~264页；2007年）。

② Truncating TLS Connections to Violate Beliefs in Web Applications，https://media.blackhat.com/us-13/US-13-Smyth-Truncating-TLS-Connections-to-Violate-Beliefs-in-Web-Applications-WP.pdf（Ben Smyth和Alfredo Pironti，Black Hat USA，2013年）。

③ Triple Handshakes and Cookie Cutters，https://www.secure-resumption.com（Bhargavan等，2014年3月）。

```
Cache-Control: no-cache, must-revalidate
Location: /account/login.html?redirected_from=/admin/
Content-Length: 0
Set-Cookie: JSESSIONID=9A83C2D6CCC2392D4C1A6C12FFFA4072; Path=/; Secure; HttpOnly
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
```

为了保护Cookie的安全，可以将Secure属性追加到标头行。但是，这个属性被放在了Cookie的名称和值的后面，如果从Path属性之后进行截断，将会得到一个不安全的Cookie。

很明显的是，如果截断了HTTP响应头，它们将变得不完整而且非法；比如结尾没有换行（CRLF）并且在整个标头的结尾没有空行。然而，现实情况显示，浏览器会忽略掉响应头中不合法的部分并继续处理它们。大多数浏览器都对一种或者几种截断攻击存在风险，详细情况如表6-4所示。

表6-4 浏览器中的TLS截断（来源：Bhargavan等）

	In-header截断	已忽略Content-Length	已忽略丢失的终止区块
Android浏览器4.2.2	是	是	是
Android Chrome 27	是	是	是
Android Chrome 28	否	否	是
Android Firefox 24	否	是	是
Safari Mobile 7.0.2	是	是	是
Opera Classic 12.1	是	是	是
IE10	否	是	是

这种攻击十分精巧，但是如果能使其过程自动化的话，也可能应用于实践中。具体可以按照以下步骤操作。

(1) *攻击一个还没有与目标网站建立会话的用户*。网站不会在旧Cookie存在的时候给用户设置新Cookie。这可以通过一些社会工程学技巧来达成，或者对于一个主动攻击者来说，就是重定向一个明文HTTP请求。

(2) *找到一个可以对HTTP响应注入任意数据的入口*。这是达成攻击的关键。这可以让你在TLS记录的边界上进行注入。例如，在很多网站上，对某些资源的访问需要通过身份验证才行，身份验证导致的重定向中包括要访问的资源的地址。你在之前应该见过这种例子，就是用redirected_from参数来达成目的。

重定向响应是一个理想的攻击发起点，因为它们没有任何内容。如果你截断的是其他响应，那么响应体的缺少会引发用户的怀疑。

(3) *插入填充数据来将响应头分割成两个TLS记录*。通常来说，HTTP的重定向响应很小并且可以装到一个TLS记录中。你的目标是将一个TLS记录分成两个。因为TLS记录最大为16 384字节，所以通过提交一个非常长的负载并且让它的长度超过这个限制，TLS栈会将HTTP响应分割成两个记录发送。

(4) *在第一个TLS记录之后关闭TLS连接*。这部分很直接：观察TLS连接并在第一个记录通过

之后立即关闭连接（例如发送RST包）。

(5) 提取不安全的Cookie。在这个环节，不完整的Cookie已经被浏览器记录，你需要做的就是从浏览器中提取它。这实际上是一种Cookie窃取攻击。

另外一个Cookie截断攻击的目标是`Strict-Transport-Security`响应头。如果你在`max-age`参数的第一个数字之后截断该响应头，则HSTS将最多在9秒后过期。此外，如果有`includeSubDomains`参数，它同样会无效。在HSTS不起作用的情况下，你可以展开HTTPS剥离攻击或者用其他方式修改Cookie，就像我们在第5章中讨论过的那样。

Cookie截断攻击被期望通过在浏览器层面进行更加严格的检查和解析来进行防御。一些厂商已经对这些问题进行了修复，但大多数厂商何时修复问题依然未知。

6.8 部署上的弱点

有些时候，不正确的部署会让系统变得脆弱，尤其是当一些广泛使用的实践方法存在漏洞的时候。本节描述的问题主要源于安全协议定义得过于抽象，没有清晰地指出服务器该如何正确地实现。结果就产生了一些微妙的问题。

6.8.1 虚拟主机混淆

一般不推荐在多域名之间共享证书，除非这些域名是紧密关联的。从某种程度上来说，这个问题的本质是因为共享证书的网站也同样在共享相同的私钥。这种共享削弱了安全性并将其降低为所有环节中最不安全的程度。同时，你并不希望多个独立的网站维护团队都能访问到同一个私钥。

然而，共享证书这种情况在应用层也存在问题，如果一个站点发生了信息泄露或者受到其他方式的攻击，其他共享相同证书的站点在合适的条件下也会遭受攻击。这些“其他的站点”可能监听的是不同的端口或IP地址并且可以从互联网上访问。

举例来说，我们假设攻击者得到了一个使用多域名证书的网站的控制权。从一个主动网络攻击行为的角度来看，攻击者发现用户连接到的其他网站也使用了同样的证书（我把这些网站称为安全站点）。然后攻击者劫持一条到这些安全站点的TLS连接并将其发往她自己控制下的那个网站。因为证书相同，受害者的浏览器不会发现任何异常并且请求会被服务器端处理。因为攻击者控制了接收请求的服务器，她可以记录那些在被劫持的连接中的Cookie并使用它们对受害者的应用层会话进行劫持。攻击者也可以对受害者的HTTP请求返回任何可以代表“安全”站点而执行的JavaScript代码。

这里有一个前提：被控制的站点的Web服务器必须忽略掉HTTP `Host`头指向某个其他网站的这种情形。取决于攻击者对服务器的控制程度，攻击者也许可以重新配置Web服务器软件来满足上述前提。但是，服务器忽略掉无效的主机信息并始终对某个默认站点作出响应的情况也是非常常见的。

Robert Hansen是第一个强调这个问题的人。当时他成功实现了一次从mxr.mozilla.org到

addons.mozilla.org的跨站点脚本攻击，这两个网站共享了同一个证书。[①]2014年，Delignat-Lavaud和Bhargavan在一篇研究报告中再次讨论了这个问题，并将其命名为虚拟主机混淆（virtual host confusion）[②]。他们同时演示了如何在几个现实场景中利用这个问题开展攻击，甚至发现了一个可能用来仿冒世界上一些最著名网站的长久安全问题。

> **注意**
> 这种攻击也可能发生在其他协议上，比如SMTP。使用同样的重定向方法，攻击者可以攻破一个脆弱的SMTP服务器然后将其他TLS连接重定向到这里。如果证书是共享的，则原本发往其他安全站点的电子邮件将被发送到攻击者这里。

6.8.2　TLS 会话缓存共享

Delignat-Lavaud和Bhargavan提及的另外一个问题是TLS会话缓存在多个不相关的网站之间共享，这种情况很常见并且容易被利用来绕过证书验证。一旦某个TLS会话建立，客户端就能与除了原始服务器之外的、但却共享同一个会话缓存的服务器恢复会话，即使该服务器并不用来处理这个请求，也没有配置正确的证书。

这个问题将所有共享同一个会话缓存的网站关联了起来（无论是基于服务器会话缓存还是基于会话票证），并且使得攻击者在攻击了一个网站后也会得到其他网站的访问权限。流量重定向（之前讨论过的伎俩）依然是这里的主要攻击手段。

对于服务器端的会话缓存，问题在于服务器没有检查当一个会话恢复的时候，它是否是再与之前建立连接的网站尝试会话恢复。会话票证的情况也同样如此。然而，对于后者，会有一些变通方法来缓解问题，就是在服务器上给每个域名独立配置票证密钥。为每个域名使用独立的票证密钥始终是一种最佳实践。

① MitM DNS Rebinding SSL/TLS Wildcards and XSS，https://web.archive.org/web/20150315042723/http://ha.ckers.org/blog/20100822/mitm-dns-rebinding-ssltls-wildcards-and-xss/（Robert Hansen，2010年8月22日）。

② Virtual Host Confusion: Weaknesses and Exploits，https://www.blackhat.com/docs/us-14/materials/us-14-Delignat-The-BEAST-Wins-Again-Why-TLS-Keeps-Failing-To-Protect-HTTP-wp.pdf（Antoine Delignat-Lavaud 和 Karthikeyan Bhargavan，2014年8月6日）。

第7章 协议攻击

多年以来，研究者一直或多或少地关注着SSL和TLS协议的安全性问题。SSL/TLS协议早期的安全性很不可靠。SSL 1存在明显的安全问题，因此Netscape于1994年年末废弃了这个版本，并发布了SSL 2来取代它。SSL 2成功地引导了电子商务的爆发，但依然不是特别安全。SSL的下一个版本SSL 3于1996年发布，解决了之前版本的许多安全问题。

在SSL 3之后，SSL/TLS协议的发展便进入了一个漫长的平和时期。1999年，SSL 3被标准化为TLS 1.0协议，它对SSL 3几乎没有修改。TLS 1.1和TLS 1.2分别在2006年和2008年发布，但是几乎所有人都还在使用TLS 1.0。2008年，人们又重新开始关注SSL/TLS协议的安全性话题。从那时起，TLS就开始不断承受彻底检查每个特性和用例细节的压力。

在本章中，我整理了近些年来针对TLS协议的攻击，关注的焦点是那些你在实际中可能遇到的问题。这些攻击按时间顺序分别是：2009年的不安全重新协商，2011年的BEAST攻击，2012年的CRIME攻击，2013年的Lucky 13、RC4偏差、TIME以及BREACH，以及2014年发现的三次握手和POODLE攻击。在本章的最后，我将简要地讨论政府部门对某些标准和加密算法进行破坏的可能性。有关FREAK和Logjam（它们是在2015年出现的，但大多不是协议问题）的信息，请参考第6章。

7.1 不安全重新协商

不安全重新协商（insecure renegotiation）也叫作TLS身份验证缺口（TLS authentication gap），是由Marsh Ray和Steve Dispensa在2009年8月首先发现的一种协议缺陷。当他们发现这个问题之后，便开始在整个业界范围内推动协议上的修复，并协调对公众披露此问题的进度。在他们还没完成上述行动的时候，Martin Rex（在同年11月）也独立发现了这个不安全重新协商的问题[①]，这使得关于漏洞的信息被过早地披露出来。[②]

---

① MITM attack on delayed TLS-client auth through renegotiation，http://www.ietf.org/mail-archive/web/tls/current/msg03928.html（Martin Rex，2009年11月4日）。

② Renegotiating TLS，http://ww1.prweb.com/prfiles/2009/11/05/104435/RenegotiatingTLS.pdf（Marsh Ray和Steve Dispensa，2009年11月4日）。

7.1.1 为什么重新协商是不安全的

重新协商的漏洞之所以存在，是因为在旧的和新的TLS连接之间没有连续性，即使这两个连接发生在同一个TCP连接上。也就是说，服务器并不会验证新旧两条TLS连接的另外一端是同一个。即使存在完整性校验，也无法保证每次重新协商后，与服务器通信的客户端都是相同的。

应用层的代码显然很少与加密层交互。例如，如果重新协商发生在HTTP请求的过程中，上层应用是得不到通知的。此外，Web服务器有时会缓存重新协商之前的数据，并将这些数据和重新协商之后的数据一并发送给上层应用。连接参数也有可能发生变化，例如一个新的客户端证书可能在重新协商之后被使用。最终的结果就是在TLS层面发生的事情与上层应用了解到的信息并不匹配。

一个中间人（man-in-the-middle，MITM）攻击者可以通过下面三个步骤来利用这个漏洞。

(1) 拦截一个受害客户端到服务器的TCP连接。

(2) 新建一个到服务器的TLS连接，包含攻击负载。

(3) 从这时起，在受害客户端和服务器之间扮演透明代理。对于客户端来说，连接刚开始，它将开始一个新的TLS握手。对于服务器来说，已经在一个已经建立的TLS连接上接收到了攻击数据，并且会将客户端的TLS握手理解为进行重新协商。一旦重新协商完成，客户端和服务器便开始交换应用层数据。攻击者的攻击负载和客户端的正常数据将会被服务器合并处理，从而使得攻击成功。

图7-1展示了攻击者是如何危害数据完整性的；这本来需要由TLS协议进行保护。攻击者可以向应用层协议的开头部分注入任何数据。这种攻击的影响主要取决于上层协议和服务器实现，具体内容在下文中讨论。

7.1.2 触发弱点

要想利用不安全重新协商进行攻击，攻击者需要找到一种能够触发重新协商的方法。在这个漏洞被发现前，大多数服务器都是允许客户端发起重新协商的，这说明这些服务器都是容易受攻击的目标。一个罕见的例外是Microsoft的IIS，它从版本6开始就不再接受客户端发起的重新协商。

但即使禁止了客户端发起的重新协商，依赖于客户端证书校验和支持SGC的网站依然容易受到攻击。攻击者只需要调查网站在哪些情况下是需要进行重新协商的。如果类似的条件得到了满足，攻击者就可以利用其展开攻击。取决于服务器端的配置，如此情况带来的攻击向量和客户端发起的重新协商有同样功效。

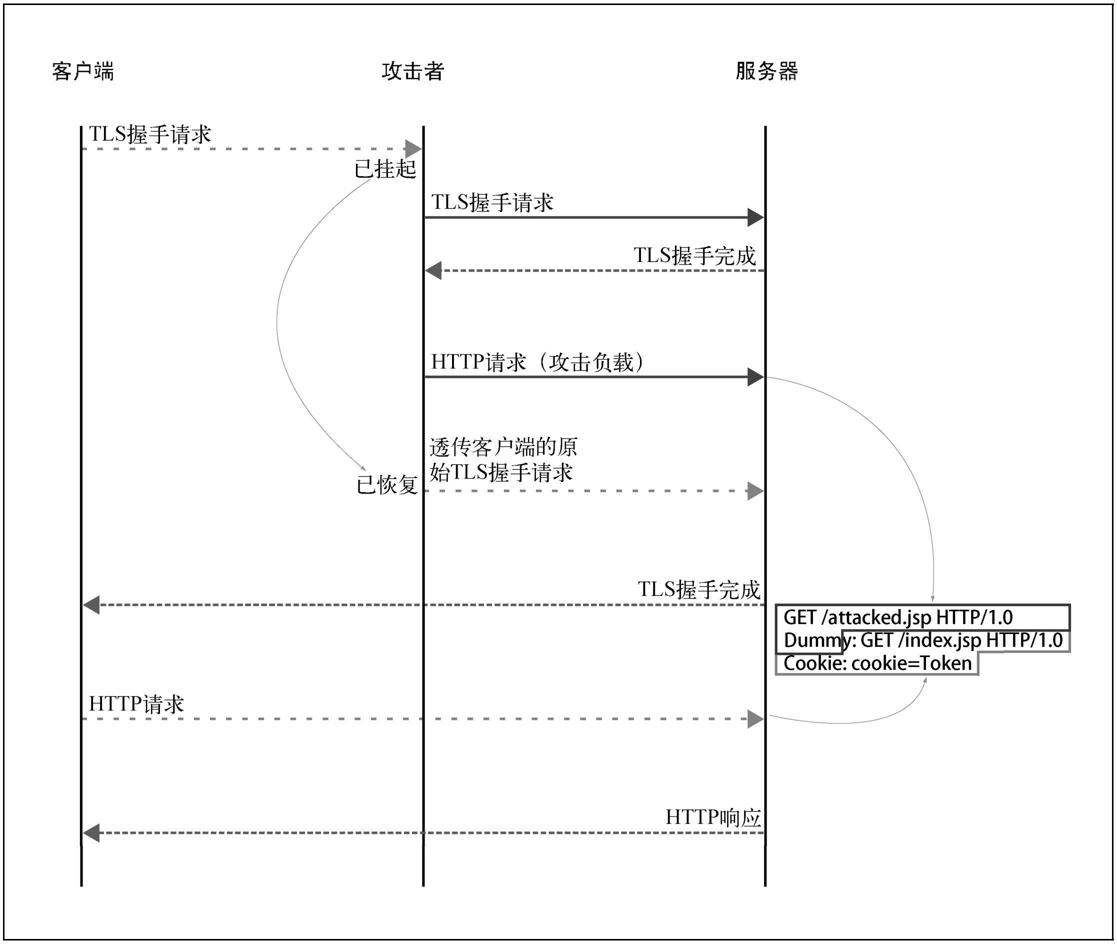


图7-1　针对不安全重新协商的中间人攻击

7.1.3　针对 HTTP 协议的攻击

基于HTTP协议来讨论不安全重新协商是最好不过的了。针对HTTP协议的攻击有很多种，它们的攻击可行性取决于目标网站的Web服务器配置和受害者的技术强度。一开始，人们只讨论了一种攻击手段，但是随后安全社区成员共同发现了其他可能的攻击方法。具体来说，Thierry Zoller花费了大量的精力来跟踪和维护攻击向量[①]，并设计用于概念验证的攻击。[②]

① 攻击向量（attack vector）是指攻击者利用某种IT系统的漏洞，发送攻击负载或者篡改响应的途径或方法，也包括人为因素。——译者注

② TLS/SSLv3 renegotiation vulnerability explained，http://www.g-sec.lu/tools.html（Thierry Zoller，2011年12月23日）。

1. 执行任意的GET请求

最容易进行的攻击就是使用受害者的凭据来执行任意的GET请求。包括攻击负载（以粗体表示）的有效请求和受害者的请求看起来可能如下所示。

```
GET /path/to/resource.jsp HTTP/1.0
X-Ignore: GET /index.jsp HTTP/1.0
Cookie: JSESSIONID=B3DF4B07AE33CA7DF207651CDB42136A
```

我们已经知道，攻击者可以在受害者的请求之前加上任意的明文。攻击者的挑战在于，如何利用这种能力来控制攻击向量，以便使真正的请求（实际上就是受害者的请求行）无效，然后利用含有关键信息（例如会话Cookie或者HTTP基本身份验证信息）的部分来成功进行身份验证。

具体来说，攻击者这样进行攻击：首先将攻击负载的开头设置成一个完整的HTTP请求行，这里将会是攻击的目标URL；然后添加一个*不完整的请求头行*（partial header line），这个请求头行是故意做成不完整形态的（结尾也没有换行），会使受害者请求的第一行无效，也就是真正的请求行。受害者请求中所有之后的请求头都会成为攻击请求的一部分。

我们从这个过程了解到了什么？攻击者可以选择将请求发到任何路径，并且随意使用受害者的身份凭据。但是攻击者无法实质上获取到受害者的身份信息，此外相应的HTTP响应也将返回给受害者。这看起来很类似于*跨站请求伪造*（cross-site request forgery，CSRF或XSRF）。大部分关心安全的网站已经处理过这个知名的Web应用安全问题，其他那些没有处理CSRF的网站更容易遭受攻击。

这就是最早展示的攻击向量；因为与CSRF相似，所以很多人忽视了此缺陷，认为其不重要。

2. 身份凭据窃取

在公开披露上述攻击方法后，改进的攻击方式开始出现。几天之后，Anil Kurmus改进了攻击方法，成功地获取到了加密数据。①

大部分人在研究可能的攻击向量时，都将注意力集中在如何利用被劫持请求中的身份凭据（例如会话Cookie或者HTTP基本身份验证信息）。Anil发现，虽然他无法直接获取到任何数据，但是仍然可以使用另外的身份将这些数据提交到网站，而此身份是由他所控制的（也可以说是反向会话劫持）。此时就只需要从网站上获取这些数据了。

他的证明概念攻击是针对Twitter进行的。他设法将受害者的身份凭据信息（在受害者的HTTP请求头中）作为自己的Twitter消息发出。下面是该请求（攻击者的负载以粗体表示）：

```
POST /statuses/update.xml HTTP/1.0
Authorization: Basic [attacker's credentials]
Content-Type: application/x-www-form-urlencoded
Content-Length: [estimated body length]

status=POST /statuses/update.xml HTTP/1.1
```

① TLS renegotiation vulnerability: definitely not a full blown MITM, yet more than just a simple CSRF，http://www.securegoose.org/2009/11/tls-renegotiation-vulnerability-cve.html（Anil Kurmus，2009年11月11日）。

```
Authorization: Basic [victim's credentials]
```

在这个改进版本的攻击中，受害者的请求被作为status参数的内容在请求体中提交。结果，Twirtter将它当成一条Twitter消息的内容，将其发送到攻击者的Twitter流中，这样攻击者就可以在自己的Twitter上看到这些信息。对于其他的网站，攻击者也许可以通过发送一个论坛帖子或者给自己发送一封邮件等类似的方法完成攻击。

这里唯一的挑战就是获得正确的Content-Length值。因为攻击者无法知道真实的长度，所以无法使用正确的长度。但是，他只需要设置一个大到可以覆盖到受害者请求中敏感数据的长度就可以成功地实施攻击。Web服务器在读取Content-Length头中所标示的长度后，就会认为请求已经结束并进行处理。受害者请求中的其他数据会被当作同一个连接上的下一个请求进行处理或者被直接丢弃（因为此时的剩余数据基本都无法构成完整的HTTP请求）。

3. 用户重定向

如果攻击者可以在一个网站上找到某个进行重定向的资源，他可能成功地执行以下几种攻击。

- **将用户重定向到恶意网站**

  网站上一个开放的重定向点，可以让攻击者将用户重定向到他选择的目的地。这对于钓鱼攻击来说非常理想，因为攻击者可以构造一个被攻击网站的复制品，同时还可能使用一个相似的域名来使得欺骗效果更好。把域名变得让人感到相似或“官方”是很简单的（例如，www.myfeistyduck.com，而真正的域名是www.feistyduck.com）。最后，为了完成钓鱼攻击，攻击者需要为恶意网站获取一个合适的证书。

- **连接降级至HTTP**

  如果攻击者可以在目标网站上找到将用户重定向到普通HTTP页面的资源，那TLS连接将会被有效地降级。之后攻击者可以使用类似sslstrip这类工具来获得对受害者浏览网站的全部控制。

- **通过重定向POST请求截获身份凭据**

  如果目标网站使用了307重定向（这要求重定向不修改原请求的方法），那就有可能将整个请求（包括POST请求体）全部发送到攻击者指定的地点。所有的浏览器都支持307重定向，不过有些浏览器会要求用户确认。[①]这种攻击是非常危险的，因为它不依赖于目标网站本身的功能。换句话来讲，就是不需要攻击者在目标网站上拥有账号。这很关键，因为真正有价值的网站可能并不容易在其上拥有账号（比如银行或者其他金融机构）。

Leviathen Security Group的论文中有关于使用重定向来进行不安全重新协商攻击的一个较深

① 在2013年7月我最后一次测试这个特性的时候，最新版本的Chrome、IE和Safari都会将请求重定向到完全不同的网站而不给出任何警告。Firefox和Opera给出了需要确认的警告，但是这些提示都可以进一步改进。例如，Firefox没有提示出请求将会被发往何处。Opera提供了大部分的信息（当前的地址以及目标地址），以及与这些信息在一起的多种处理选项，包括取消，对POST方法继续处理或者转变成GET方法等。这些选择对普通用户来说依然过于令人迷惑。

入的探讨。[①]

4. 跨站点脚本

在某些很少见的情况下，攻击者可能会向受害者的浏览器中注入HTML和JavaScript脚本来实现通过*跨站点脚本*（cross-site scripting，XSS）对其的完全控制。这有可能通过使用HTTP的`TRACE`方法来实现，该方法要求服务器在响应中镜像HTTP请求。在受攻击的情况下，镜像的请求中将带有攻击负载。

这种攻击方法对主流的浏览器不起作用，因为`TRACE`响应的内容类型是`message/http`。但是，Thierry Zoller指出，有一些不太常用的Windows浏览器总是将响应当作HTML解析，这些浏览器是存在风险的。此外，定制化的脚本也缺少对响应类型的检查，它们也许同样存在风险。

7.1.4　针对其他协议的攻击

虽然针对HTTP协议的攻击吸引了人们大部分注意力，但是我们可以假设所有基于TLS的协议都有可能遭受不安全重新协商的威胁。任何在重新协商前后不重置会话状态的协议都存在风险。

- SMTP

  Wietse Venema是Postfix项目的一名成员，他发表了一篇关于不安全重新协商对SMTP和Postfix邮件服务器的影响的分析报告[②]。该报告指出，SMTP协议是存在漏洞的，但是却很难利用其漏洞展开攻击。这是因为不像HTTP协议，一个SMTP事务由多个命令和响应组成。他总结到，Postfix不存在风险，但这仅仅是运气比较好，因为某些设计上的选择使得攻击无法展开。报告同样给出了一些改进客户端和服务器端软件的方法来实现对不安全重新协商的防御。

  不安全重新协商之所以没有对SMTP产生巨大的威胁，其根本原因是大多数SMTP服务器都不使用有效的证书，这导致了大多数客户端不去校验证书的有效性。换句话说，出于这个原因，针对SMTP的中间人攻击已经是非常简单的了，攻击者也犯不着使用更难的方法进行攻击了。

- FTPS

  Alun Jones是WFTPD的作者，她发表了一篇关于不安全重新协商对FTPS的影响的分析报告[③]。其主要结论是，由于在某些FTP服务器中文件传输的实现问题，导致中间人攻击者可以利用不安全重新协商来告知服务器关闭命令通道的加密功能。这样，传输的文件的完整性就可能遭受攻击。

---

① Generalization of the TLS Renegotiation Flaw Using HTTP 300 Redirection to Effect Cryptographic Downgrade Attacks，http://www.leviathansecurity.com/white-papers/tls-and-ssl-man-in-the-middle-vulnerability/（Frank Heidt和Mikhail Davidov，2009年12月）。

② Redirecting and modifying SMTP mail with TLS session renegotiation attacks，http://www.postfix.org/wip.html（Wietse Venema，2009年11月8日）。

③ My take on the SSL MITM Attacks – part 3 – the FTPS attacks，http://blogs.msmvps.com/alunj/2009/11/18/my-take-on-the-ssl-mitm-attacks-part-3-the-ftps-attacks/（Alun Jones，Tales from the Crypto，2009年11月18日）。

7.1.5　由架构引入的不安全重新协商问题

有时候系统设计和架构决策可能会导致本不应该存在的重新协商问题的发生。我们用SSL卸载来举例。这个实践经常用于给本不支持添加加密的服务添加加密，或者用于通过从主服务点移除TLS处理来提高系统的性能。如果在TLS终止点处支持不安全的重新协商，那么即使真正的Web服务器很安全，整个系统也存在被攻击的风险。

7.1.6　影响

不安全重新协商是一个严重的威胁，因为它完全破坏了TLS本该拥有的安全性。不仅通信的完整性遭受了攻击，攻击者还可能会获取到通信内容。同时诞生了很多种不同的攻击手段，从CSRF到凭据窃取再到充满欺骗性的钓鱼攻击。因为这种攻击的实施要求攻击者具有良好的技术背景和充分的目标网站调查，所以进行这类攻击的攻击者大多是针对高价值网站的主动攻击者。

对于攻击者来说，一个理想的场景是针对自动化系统的攻击，这种系统很少检查错误，而且没有记录足够的日志，此外还会不停地重试请求直到请求成功为止。这个场景极大地扩大了攻击面，并且比直接攻击最终用户（浏览器）更加容易。

利用不安全重新协商进行攻击的原理已经广为人知，并且相关的工具也可以轻易获得。针对Twitter的概念验证攻击方法可以在互联网上找到，只要对任何中间人攻击工具进行小幅修改，就可以实现该攻击方法。

对数据完整性的破坏带来了另外的边界效应，这源于攻击者可以使用受害者身份提交任意数据这个事实。即使攻击者无法获取任何数据或者欺骗受害者，他仍然可以伪造攻击负载以达到受害者对服务器进行攻击的效果。因为大部分网站的日志记录不足，导致这种攻击（使用受害者身份）几乎无法阻止，并对受害者造成灾难性的后果。因此，最终用户应该将他们的浏览器设置为只与支持安全重新协商的服务器通信。[①]

7.1.7　缓解方法

有几种不同的方法可用来处理不安全重新协商问题，其中有一些方法是比较合理的。

- 升级以支持安全的重新协商

  2010年早期，在协议层引入了Renegotiation Indication扩展以便解决不安全重新协商的问题。[②]几年之后的今天，你可以认为所有的产品都可以进行升级来支持安全重新协商。如果你面临的是正在使用某些无法升级的产品，那么可能需要考虑是否还继续使用它们。

- 禁用重新协商

  在漏洞被发现的头几个月里，唯一可行的有效缓解方法就是禁用重新协商功能。

  这种方法比支持安全重新协商要差。首先，某些场景确实需要重新协商功能（例如在需

① 例如，在Firefox的about:config页面中，将security.ssl.require_safe_negotiation设置更改为true。

② RFC 5746: TLS Renegotiation Indication Extension，https://tools.ietf.org/html/rfc5746（Rescorla等，2010年2月）。

要对客户端证书进行验证的时候)。其次，不支持安全重新协商促进了网络上的重新协商的不确定性，使得用户无法有效地保护自己。

禁用SSL重新协商是权宜之计而非解决方法

我们所有人都应该努力推动升级我们的系统来支持安全重新协商。在2009年或2010年，如果你在系统中关闭重新协商，可能觉得这很安全并且不需要采取进一步的措施。从一个很狭义的角度来说，这是正确的。然而，不支持安全重新协商会造成重大的倒退，因为这阻止了浏览器采用更加严格的重新协商策略。

不像服务器，要么就是要求重新协商，要么就是接收到对端主动提供的重新协商请求，在发生攻击时，浏览器无法发现重新协商是否正在发生。毕竟，浏览器并没有参与到重新协商的过程中。

对于浏览器来说，唯一可行的保护自己的方法，就是拒绝与不支持安全重新协商的服务器建立连接。那么问题就产生了：在互联网上还存在着大量的这类服务器，并且任何一个浏览器都不希望自己无法连接上这些网站。服务器禁用重新协商也许对于自己来说是安全的，但是这种行为显著拖延了问题的解决周期，因为这种行为增加了无法确定为安全的服务器的总数。

7.1.8　漏洞发现和补救时间表

不安全重新协商漏洞给了我们一个少有的机会来检查和评估我们修复一个协议漏洞的能力。很明显，在一个像TLS这样复杂的生态系统中，修复任何问题都需要大量的协作，并要花费数年时间。但具体多少年呢？图7-2给了我们一个很好的参考。

在这个时间轴上我们需要的东西大概包括以下各项。

(1) 大约6个月来修复协议。

(2) 在这之后的12个月用来对库和操作系统进行修复和打补丁。

(3) 再之后的24个月用来对其他大多数系统进行修复（或者下线老旧系统）。

根据Opera完成的评估来看，他们跟踪的系统中有大约50%在官方RFC发布之后的一年内，支持了安全重新协商。[①]基于同样的调查数据，2014年2月，这一数字为83.3%[②]。结论就是我们大概需要4年左右的时候才能处理完这种类型的缺陷。

当我写下这些文字的时候，也就是2014年的6月，互联网上88.4%的服务器支持了安全重新协商，数据来自SSL Pulse数据库。[③]大约6.1%的服务器还在使用不安全重新协商，约6.8%的服务器完全不支持重新协商。各种比例的服务器加和之后的总比例超过了100%，这是因为有1.3%的服

① Secure browsing like it’s 1995，https://web.archive.org/web/20140302021900/http://my.opera.com/securitygroup/blog/2011/03/17/secure-browsing-like-its-1995（Audun Mathias Øygard，2011年3月17日）。

② Re: Call for acceptance of draft-moeller-tls-downgrade-scsv，http://www.ietf.org/mail-archive/web/tls/current/msg11270.html（Yngve N. Pettersen，2014年2月9日）。

③ SSL Pulse，https://www.trustworthyinternet.org/ssl-pulse/（SSL Labs，检索于2014年7月15日）。

务器同时支持了安全的和不安全的重新协商。

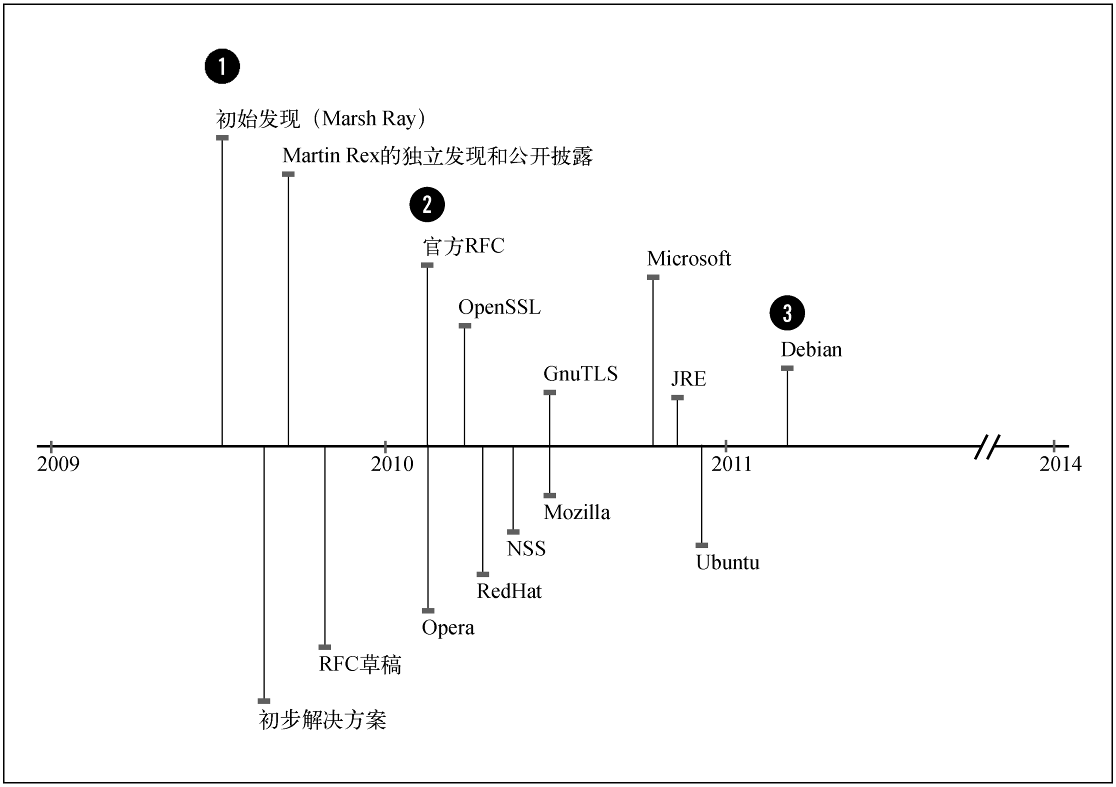


图7-2　不安全的重新协商修复时间轴

7.2　BEAST

2010年夏天，Duong和Rizzo公布了一种新的攻击方法，可以针对TLS 1.0以及更早的协议使用，最终可以获取到少量的加密数据的内容。[①]这种攻击建立在TLS 1.0协议中已知的可预测的初始向量问题之上。这个弱点已在TLS 1.1中修复，因此被认为无法利用其进行攻击；但是在发现此攻击的时间点上，几乎没有浏览器支持新版本的TLS协议。

很多年里，这个所谓的BEAST攻击为整个生态系统敲响了警钟。第一，它（再次）强调了攻击发展得越来越完善。在本节后面你会了解到，这是一个被忽略了将近10年之久的潜在问题，却在两个积极主动的研究者手里变成了切实可行的攻击手段。Duong和Rizzo教育我们不应当忽略小问题，因为任何小问题都可能最终变成大问题。

① Here come the ⊕ Ninjas，http://www.hit.bme.hu/%7Ebuttyan/courses/EIT-SEC/abib/04-TLS/BEAST.pdf（Duong和Rizzo，不完全版本，2011年6月21日）。

第二，对攻击的披露以及造成的焦虑显示出浏览器厂商十分不关注TLS协议。他们与业界的其他软件厂商一样，投入了太多的精力来关注漏洞的可利用性，而对需要花费很多年，通过大量的客户端和服务器通力协作才能修复的协议问题以及其他类似问题，则根本不去关心。这种类型的问题和缓冲区溢出这种缺陷相比是截然不同的，后者可以相对快速地进行修复，而前者不能。

Thai在博客上坦率地发表了关于BEAST的问题。[①]当他没有能够吸引起浏览器厂商足够的注意时，你能感受到他的沮丧，这是因为虽然他能够在一个模拟的环境中演示攻击，但是却无法在实际环境中演示。不过他们没有放弃，而是继续构建有效的概念证明环境，成功地演示攻击，并最终引起了业界足够的重视。

7.2.1　BEAST 的原理

BEAST攻击针对的是TLS 1.0和更早版本的协议中对称加密算法的CBC模式。像之前提到的那样，主要问题是IV可以预测，这就使得攻击者可以有效地将CBC模式削弱为ECB（electronic code book，电子密码本）模式，而ECB模式则是不安全的。

1. ECB Oracle

ECB是最简单的操作模式：它将数据分割成固定大小的块并分别进行加密。这种方法有几个安全问题，不过我们现在比较关心的是ECB不会改变块加密算法的确定性本质。也就是说，如果你加密的是相同的数据块，则加密后的结果也是相同的。这对于攻击者来说是一个非常有用的性质：如果攻击者能够提交任意数据进行加密，就可以通过猜测获取之前加密数据的内容。流程如下所示。

(1) 截获一段加密数据，数据的大小取决于加密算法，例如AES-128是16字节。

(2) 对16字节的明文进行加密。因为块加密的特性（输入中任一位的变化将导致整个输出的变化），所以攻击者每次都猜测整块数据。

(3) 将加密后的数据与第一步中截获到的加密数据对比，如果两者一致，则猜测正确，否则继续执行第二步。

因为攻击者每次只能猜测整块数据，所以这不是一个好的攻击方法。为了猜测16字节的数据，攻击者需要进行2^{128}次猜测，平均也要2^{127}次猜测。我们后面会看到，总会有办法来改进攻击效率。

2. 可预测IV的CBC

CBC模式和ECB模式的最大区别是，CBC使用了一个*初始向量*（initialization vector，IV）来在加密之前掩码明文，这样做的主要目的是隐藏最终密文中的模式规律。采用合理的掩码手段，即使在明文相同的情况下，加密后的密文也会完全不同。因此，CBC并不像ECB那样受到明文猜测的威胁。

为了让IV切实有效，必须使IV在每条消息上都不可预测。一种实现方法是为每个要加密的数据块都准备一个随机数。但是这不太实际，因为这会让我们的输出变成2倍大。在实际中，SSL 3和TLS 1.0中的CBC模式只会对待加密数据的开头使用一个随机数，之后当前数据块加密之后的

① BEAST，http://vnhacker.blogspot.co.uk/2011/09/beast.html（Thai Duong，2011年9月5日）。

密文会作为下一个数据块的IV，因此我们把这种情况叫作“链接”（chaining）。

这种“链接”的方法，只有在攻击者无法看到加密数据以及无法影响下一个加密块内容的情况下才是安全的。否则，简单地通过查看加密数据就可以知道每个加密块的IV是什么。不幸的是，TLS 1.0以及更早的版本将整个连接当作一个消息处理，并只为这条消息的第一个TLS记录分配一个随机IV，所有后续的TLS记录使用前一个加密块作为自己的IV。由于攻击者可以看到全部加密数据，他就可以了解到从第二个记录开始的每个记录的IV。TLS 1.1和TLS 1.2使用基于每个记录的IV，因此没有这种问题。

TLS 1.0的这种方式在面对一个可以提交任意明文数据进行对称加密的主动攻击者来说，后果是灾难性的，攻击者可以基于观察到的密文的特征，对攻击手段进行改造。也就是说，协议可能会受到一种基于分块的明文选择攻击。在IV可以预测的情况下，CBC就会降级成ECB。

图7-3用三个数据块来展示这种攻击：两个数据块是浏览器发送的，一个是攻击者（通过浏览器）发送的。为了简化描述，我让每个数据块都对应一个加密块，并且没有考虑填充的问题，虽然TLS协议同样会做填充。

攻击者的目的是要获得第二个数据块的内容。他没有办法把第一个数据块作为目标，这是因为这个数据块的IV永远也不会出现在网络上。但是攻击者在看到第一个加密的数据块后，就知道了第二个数据块的IV（IV_2），同样也可以得知第三个数据块的IV（IV_3），他自然也知道第二个加密的数据块（C_2）。

在看到网络上的前两个数据块后，攻击者接管用户的浏览器并让其提交攻击者制定的明文进行加密后的数据。对于每次猜测，攻击者都可以观察网络上的加密数据。因为他知道所有的IV，因此攻击者可以将用于猜测的数据制造成特殊的形态来使IV无效。当猜测成功后，猜测的密文（C_3）应当与被猜中的密文（C_2）一致。

要了解如何可以有效地清除IV，我们必须看一些数学上的东西。我们来看明文M_2，其中有一些攻击者要了解的信息，以及M_3，就是攻击者制造的猜测明文：

$$C_2 = E(M_2 \oplus IV_2) = E(M_2 \oplus C_1)$$
$$C_3 = E(M_3 \oplus IV_3) = E(M_3 \oplus C_2)$$

明文首先与它们的IV进行异或，然后再进行加密。因为使用的是不同的IV，如果M_2和M_3相同，则每次加密后的密文都会不同。但是，因为我们知道所有的IV（C_1和C_2），我们就可以用一种方式来抵消掉掩码数据的效果。假设M_g是我们要生成的猜测结果：

$$M_3 = M_g \oplus C_1 \oplus C_2$$

那么对M_3进行加密将会是这样：

$$C_3 = E(M_3 \oplus C_2) = E(M_g \oplus C_1 \oplus C_2 \oplus C_2) = E(M_g \oplus C_1)$$

那么如果我们的猜测是对的，那么我们猜测数据的密文C_3将与第二个数据块的密文C_2相等：

$$C_3 = E(M_g \oplus C_1) = E(M_2 \oplus C_1) = C_2$$

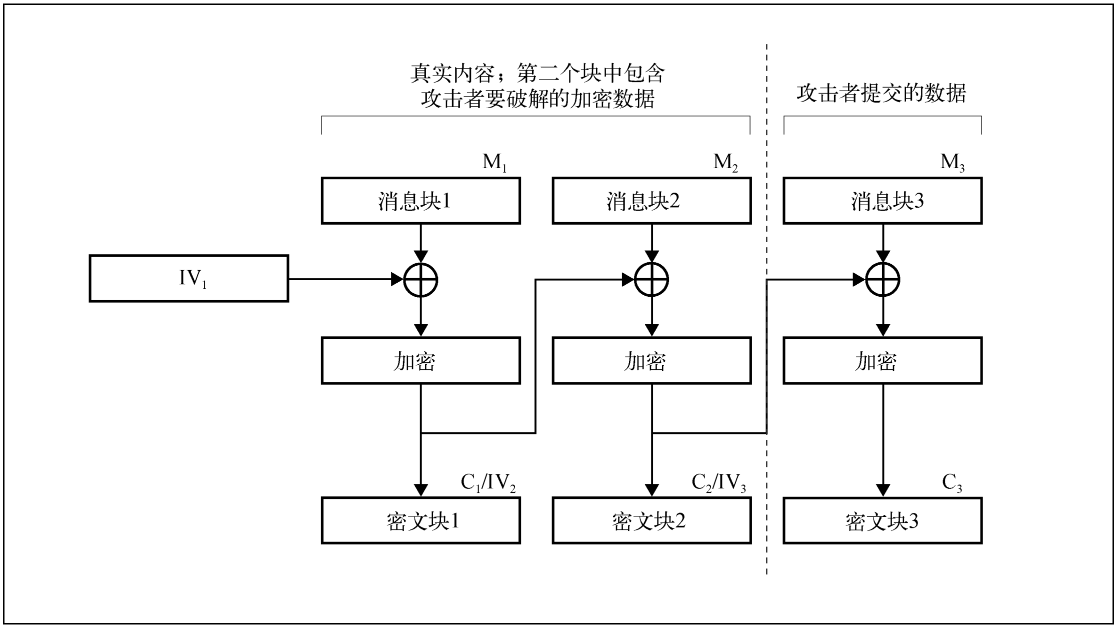


图7-3　针对可预测IV的CBC模式的BEAST攻击

3. 实际攻击

虽然我们知道了可预测IV问题的具体情况，但是由于我们需要每次猜测至少一个整块的数据（一般为16字节），展开攻击依然比较困难。然而，对于HTTP协议，我们可以做一些优化。

- ❑ HTTP协议通常含有小段的敏感数据，例如密码和会话令牌。因此有时猜测16字节就已经足够了。
- ❑ 敏感数据通常使用有限的字符集，例如会话令牌通常都是以十六进制数字的形式存在，这说明了其中只有16种字符的可能。
- ❑ HTTP消息的结果是非常容易预测的，这就说明了敏感信息存在于我们熟悉的数据中。例如，字符串`Cookie:`会永远存在于HTTP请求中第一个Cookie的名称之前。

将上面这些因素考虑在内后，展开攻击的必要猜测次数显著地减少了，但是依然没有减少到可以用于实际操作的数量。

BEAST真正成为可实施的攻击，是在Duong和Rizzo发现了现代浏览器可能会被训练有素的攻击者操控，获得空前的控制权力之后。决定性的条件是，攻击者可以：(1) 影响敏感信息在HTTP请求中的位置；(2) 对加密内容及其发送时间进行完全控制。

第一个条件不难满足，例如要想将Cookie修改到指定位置，只需要在请求行上加入一些特定内容即可。第二个条件则存在一些问题，这种级别的控制无法通过JavaScript满足。但是Duong和

Rizzo发现可以使用Java applet。他们还需要利用一个独立bug来让Java将流量发送到任意网站。[1]他们需要这样做以使得BEAST攻击更加通用并可以用来攻击任何网站。利用这个Java中的额外问题并非总是必须的。可能可以利用允许用户上传内容的网站来上传Java applet。之后他们在目标网站环境中运行这些Java applet并控制浏览器发送数据。[2]

还有另外一个条件需要满足，就是需要能观察到加密流量，只有这样才能决定下一个IV。此外，这些IV需要由浏览器中的Java程序发送。

实际中，BEAST是一种主动的网络攻击。虽然可以使用社会工程学的技巧来让受害者访问含有恶意JavaScript脚本的网站，但是更简单的方式是针对受害者访问的明文网站进行劫持，然后将恶意脚本注入到响应中。

如果你能达成上述所有条件，那实施BEAST攻击是很容易的。通过改变敏感数据在HTTP请求中的内容，你可以构造出一个含有15字节的已知内容数据和一个1字节敏感数据内容的加密块。猜测1字节是比较容易的，你只需要最多猜测2^8（256）次，平均2^7（128）次。假设在低熵的情况下（例如十六进制数字），你可以对于每个字符低到猜测8次（平均）；如果对计算时间要求较高，还可以并行提交多次猜测。

7

恶意JavaScirpt脚本

恶意JavaScript脚本泛指那些在受害者浏览器中运行的恶意代码。大多数恶意脚本是用来攻击浏览器本身以达到仿冒用户或者攻击其他网站的目的。BEAST第一个利用恶意JavaScript脚本来针对密码领域进行攻击，并在之后被大量模仿。你可以在本章稍后部分找到相关细节。

对于恶意JavaScript脚本的使用是改变安全模型的一个很好的例子。在1994年SSL协议最早被设计出来的时候，浏览器还只能进行简单的HTML渲染。今天，它们已经成为了强大的应用交付平台。

7.2.2 客户端缓解方法

BEAST是发生在客户端的一种缺陷，因此需要将解决问题的措施部署在客户端。在2004年，该问题首次被发现之后，OpenSSL尝试通过在每个真TLS记录前插入一个空TLS记录的方式来解决此问题。在这种方法下，虽然攻击者可以预测到下一个IV，但是这个IV是给没有内容的0长度TLS记录使用的。有效的数据在之后的TLS记录中传输，但是这个记录使用了攻击者无法进一步得知的IV，这就说明了无法进行攻击。

---

① 在没有获得许可的情况下，Java Applet与它们的父网站通信。这个限制也叫作同源策略（same-origin policy，SOP）。Duong和Rizzo发现了一种绕过此限制的方法。这种绕过方法目前是否还有效尚不清楚，我参考了2013年发行的Java版本，这个问题依然还存在，不过需要额外的一些工作来触发它。

② The pitfalls of allowing file uploads on your website，http://labs.detectify.com/post/86302927946/the-lesser-known-pitfalls-of-allowing-file-uploads（Mathias Karlsson和Frans Rosén，2014年5月20日）。

不幸的是，这个方法在实际中没用，这是因为有些浏览器对0长度TLS记录支持得十分不好。因为在当时并没有实际的BEAST攻击产生，因此OpenSSL放弃了这个修改，到目前为止，据我们所知，还没有任何一个SSL库解决了这个问题。

在2011年开始，浏览器使用了一种空记录技术的变种来抵御BEAST攻击。该方法叫作1/*n* – 1分割，由Xuelei Fan提出。①该方法仍然是发送两个TLS记录，但是将应用层数据的第一字节放在了第一个记录中，其余的字节放在第二个记录中。这个方法为数据创造了有效的随机IV：第二个记录中的数据是安全的。整个数据的第一字节仍然使用了可预测的IV，但是因为这字节所在的加密块中，至少存在随机的、并且每个记录都不相同的其他7字节（更常见的是15字节），所以攻击者无法很容易地猜测到这些字节。

1/*n* – 1分割要比原始的防御方法更好，但是其部署过程依然不太顺利。Chrome最早实现了这个方法，但是很快就不得不回退掉这个修改，因为太多的网站不兼容而导致无法访问。②但是由于Chrome浏览器的开发者坚持这个修改，不久之后其他的浏览器厂商也加入到其中，使得修复可以推广开。

1/*n* – 1分割的开销在于，每次客户端发送数据的时候，都有额外的37字节要发送。③

表7-1显示了在主流平台上的BEAST修复进度。

表7-1　主流库、平台和浏览器的BEAST缓解进展

产　　品	版本（时间）	注　　释
Apple	OS X v10.9 Mavericks（2013年10月22日）和 v10.8.5 Mountain Lion（2014年2月25日）	1/*n* – 1分割从10.8.5版本开始提供，不过默认是禁用的。原本的设计是可以对此功能进行配置，但是一个bug导致了默认值无法被修改[a]
Chrome	v16（2011年12月16日）	最早在v15中引入，但是进行了回退，因为很多网站无法正常浏览
Firefox	v10（2012年1月31日）	在v9的时候就计划开发，但是Mozilla在最后一刻改了主意，主要是为了给不兼容的网站更多的时间进行升级[b]
Microsoft	MS12-006[c]（2012年1月10日）	该缓解手段在IE浏览器中启用，但是其他使用Schannel（Microsoft的TLS库）的软件则默认没有启用。Microsoft推荐在没有浏览器的场景下使用TLS 1.1来解决BEAST问题。Microsoft知识库的2643584文章详细地讨论了相关配置[d]
NSS	v3.13[e]（2011年10月14日）	对所有应用程序默认启用
OpenSSL	尚未缓解	此问题由bug #2635跟踪[f]

① Bug #665814, comment #59: Rizzo/Duong chosen plaintext attack (BEAST) on SSL/TLS 1.0，https://bugzilla.mozilla.org/show_bug.cgi?id=665814#c59（Xuelei Fan，2011年7月20日）。

② BEAST followup，https://www.imperialviolet.org/2012/01/15/beastfollowup.html（Adam Langley，2012年1月15日）。

③ 某些客户端（例如Java和OS X）在第一组的时候并没有采用防御BEAST的对策，它们在第二组之后才开始实施。这节省了带宽但是降低了安全性。应用层数据可能依然是安全的，因为如果要实施一次猜测，你至少要先看到加密的数据。然而，在任何应用层数据被发送之前，TLS需要以自己的目的使用加密，也就是Finished消息，该消息可能不是那么引人注意，因为每次连接该消息都发生变化。但是，TLS协议在进化，其他的位和字节在第一条消息中也可能被加密。理论上，未来的变化可能使得TLS更加容易受到攻击。在实际中，BEAST攻击在TLS 1.1中被修复，因此TLS 1.0的服务器中是很少有支持这些新特性的。在TLS 1.1中，新增加的开销与一个加密块的大小相等，一般是16字节。

（续）

产　品	版本（时间）	注　释
Opera	v11.60[g]（2011年12月6日）	“修改了一个低严重级别的问题，该问题由Thai Duong和Juliano Rizzo发现；细节将在随后公布”出现在v11.51的Release Note中，但是之后该说明却被删除了
Oracle	JDK 6u28 and 7u1（2011年10月18日）[h]	

a Apple enabled BEAST mitigations in OS X 10.9 Mavericks，http://blog.ivanristic.com/2013/10/apple-enabled-beast-mitigations-in-mavericks.html（Ivan Ristić，2013年10月31日）。

b Bug #702111: Servers intolerant to 1/n-1 record splitting. “The connection was reset”，https://bugzilla.mozilla.org/show_bug.cgi?id=702111 （Bugzilla@Mozilla，2011年11月13日）。

c Microsoft Security Bulletin MS12-006，https://technet.microsoft.com/library/security/ms12-006（2012年1月10日）。

d Microsoft Knowledge Base Article 2643584，https://support.microsoft.com/en-us/kb/2643584（2012年1月10日）。

e NSS 3.13 Release Notes，https://groups.google.com/forum/#!msg/mozilla.dev.tech.crypto/KMn9LWX3vVU/-XKglCaGHLcJ（2011年10月14日）。

f Bug #2635: 1/n-1 record splitting technique for CVE-2011-3389，https://rt.openssl.org/Ticket/Display.html?id=2635&user=guest&pass=guest（OpenSSL bug tracker，创建于2011年10月）。

g Opera 11.60 for Windows changelog，http://www.opera.com/docs/changelogs/windows/1160/（2012年12月6日）。

h Oracle Java SE Critical Patch Update Advisory - October 2011，http://www.oracle.com/technetwork/topics/security/javacpuoct2011-443431.html（Oracle公司的网站）。

很多客户端工具（例如，库和命令行程序）依然没有使用$1/n-1$方法，因此理论上存在风险，但是它们不太容易被攻击。在没有办法向通信注入任意明文的情况下，攻击者无法利用此漏洞展开攻击。

7.2.3 服务器端缓解方法

虽然BEAST可以在客户端解决，但是我们依然无法控制数量巨大的浏览器的升级过程。但是因为Chrome及其自动更新的出现，使得事情好了很多。Firefox现在也在使用同样的策略，并且Microsoft可能也将会采取这种策略。但是依然有大量存在漏洞的浏览器没有修复。

直到2013年，在服务器端抵御BEAST的较好的方法是使用RC4算法。因为不会再有CBC模式出现，所以也就没有受BEAST攻击的可能。但是在2013年早期我们发现了两种新的攻击，一个是针对RC4的，另外一个则针对TLS的CBC构建（在本章稍后部分将详细介绍这两种攻击）。RC4暴露出来的问题使得我们无法继续在服务器端使用其来抵御BEAST攻击。

现在摆在我们面前的问题是，让用户要么遭受BEAST的威胁，要么遭受RC4漏洞的威胁。因为这两种攻击在实际中都不太可能发生，所以选择起来比较困难。在这种情况下，我们就需要考虑得更长远一些。如果你能找到满足之前那些条件的受害网站，才能进行BEAST攻击，这说明了大规模地进行BEAST攻击是不可能的。BEAST更适合对有针对性的用户进行攻击，并且还得假设用户使用了没有打补丁的软件并且启用了Java。但是整体的攻击成功率会很低。更重要的是，随着时间的推移，攻击成功的可能性会越来越小。

7.2.4　历史

可预测IV问题的历史，最早要追溯到1995年，当时Phil Rogaway发表了一篇针对IPsec标准的草稿[①]，有关密码学相关构造的评论文章。他指出：

> ……让攻击者无法预测到IV是必要的。

很明显，该问题没有被广泛理解，证据就是可预测的IV出现在了SSL 3（1996年）以及 TLS 1.0（1999年）的协议中。

2002年，该问题在SSH协议[②]中被再次发现并同时发现可以用于TLS协议[③]。使用空TLS记录的解决方法在2002年5月的时候引入OpenSSL，但是仅仅两个月之后就被禁用，因为这种方法引起了很多兼容性问题：包括IE在内的一些浏览器无法正确处理空TLS记录。[④]

显然，当时没有人认为这个问题值得继续跟进，因此也就没有人来寻找抵御攻击的方法。这几乎在真正的攻击被发现之前，浪费了几乎10年的时间。依然，那年有两篇论文：一个是讨论如何修复SSH协议[⑤]，另外一个是讨论关于分块适应性攻击的话题，这其中就涉及CBC模式[⑥]。

2004年，Gregory Bard展示了TLS中的可预测IV是如何被利用来获取敏感数据的。[⑦]他指出了SSL 3.0和TLS 1.0中CBC模式加密的内在问题：

> 我们演示了：该问题在SSL中引入了一个漏洞，这个漏洞可以很容易地将像密码或者PIN码这种低熵的加密字符串进行恢复。此外，我们认为，Web浏览器的开放天性使得通过有问题的插件而形成攻击的注入点是可行的……

Bard没有找到一种可以实际进行攻击的方法，但是不久便发表了另外一篇论文[⑧]，讨论了SSL上的分块适应性明文选择攻击。论文中说明了敏感数据在加密块中的位置会极大地影响要恢复密文的猜测次数。

---

① Problems with Proposed IP Cryptography，http://web.cs.ucdavis.edu/~rogaway/papers/draft-rogaway-ipsec-comments-00.txt（Phil Rogaway，1995年4月3日）。

② An Attack Against SSH2 Protocol，http://www.weidai.com/ssh2-attack.txt（Wei Dai，2002年2月6日）。

③ Re: an attack against SSH2 protocol，http://www.mail-archive.com/openssl-dev@openssl.org/msg10664.html（Bodo Moeller，2002年2月8日）。

④ 不过即使这种对策依然保留，可能也无法解决BEAST攻击。TLS是一种双工协议，有着两个独立的数据流，一个由客户端发出，另一个由服务器发出，两者使用单独的IV。在服务器上实现的空片段缓解技术无法修复客户端流中的相同漏洞，而客户端流中的漏洞正是BEAST攻击的重点。浏览器使用的TLS库（例如NSS和SChannel）没有针对可预测IV的防御方法。

⑤ Breaking and Provably Repairing the SSH Authenticated Encryption Scheme: A Case Study of the Encode-then-Encrypt-and-MAC Paradigm，https://eprint.iacr.org/2002/078.pdf（Bellare、Kohno和Namprempre，Ninth ACM Conference on Computer and Communication Security，2002年11月18日）。

⑥ Blockwise-Adaptive Attackers: Revisiting the (In)Security of Some Provably Secure Encryption Modes: CBC, GEM, IACBC，http://link.springer.com/content/pdf/10.1007%2F3-540-45708-9_2.pdf（Joux、Martinet和Valette，第17~30页，CRYPTO 2002）。

⑦ Vulnerability of SSL to Chosen-Plaintext Attack，http://eprint.iacr.org/2004/111（Gregory V. Bard，ESORICS， 2004年）。

⑧ A Challenging but Feasible Blockwise-Adaptive Chosen-Plaintext Attack on SSL，https://eprint.iacr.org/2006/136.pdf（Gregory Bard，SECRYPT，2006年）。

这个协议上的漏洞最终在2006年的TLS 1.1协议中被修复，采用的方法是为每个TLS记录设置一个随机IV。然而，修复协议并没有收到良好的效果，因为大多数浏览器都没有实现新的协议。只有在2011年BEAST漏洞被发现之后，浏览器厂商才开始考虑支持新的协议。

2011年，大部分库和浏览器厂商实现了1/*n* – 1分割缓解技术。与花费大量时间进行问题研究相比，修复其实很简单：对于NSS，只改动了30行代码。[①]

Apple在直到2013年才在他们的TLS库中实现了防御BEAST的技术（也包括Safari）。对于协议的支持，直到2013年底主流浏览器才都开始默认支持TLS 1.2。

7.2.5 影响

如果BEAST攻击成功之后，攻击者可以获得受害者的会话凭据，攻击者可以使用此凭据来访问整个目标网站。基于此，他可以在目标网站上以受害者的身份执行任意操作。在正确的条件下，BEAST容易发起，但是实际上很难满足这样的条件，尤其是在当下。

因为产生BEAST攻击的本质是协议的漏洞，所以在攻击被确认的时候，几乎所有的SSL和TLS客户端都存在风险。BEAST是客户端才有的漏洞。TLS有两条数据流，一条为客户端到服务器，另外一条是服务器到客户端。BEAST攻击瞄准的是客户端的数据流并且要求攻击者可以精确控制发向服务器的数据。这种交互能力非常关键，如果没有这种能力，攻击就无法成功。虽然服务器端的数据流也存在同样的问题，但是由于攻击者无法控制服务器的响应数据，在服务器端进行BEAST攻击没有可能。

除了可交互的需求之外，还需要满足以下两个额外的服务器控制的条件。

- CBC模式的套件需要优先

  因为只有CBC模式的套件才会有问题，所以如果服务器配置的密码套件中的RC4优先级比CBC高的话（或者根本不支持CBC），则BEAST攻击无法开展。即使客户端和服务器端都支持CBC模式的套件，攻击者也无法控制套件的选择。

- 禁用TLS压缩

  TLS可以在加密之前对内容进行压缩。压缩并不能防止BEAST攻击，但是却能使其更加困难。正常来讲，攻击者发送的数据是加密传输的。在启用压缩的情况下，内容首先被压缩，这意味着攻击者不知道压缩后的内容是什么。为了让攻击生效，攻击者需要猜测压缩后的数据，这是十分困难的。因此，Duong和Rizzo实现的BEAST攻击原始版本无法针对进行压缩的TLS数据展开攻击。在我的评估中，在BEAST攻击被发现的时候，互联网上大概有一半的服务器启用了TLS压缩。然而，客户端支持TLS进行压缩的比例在那时比较低，时至今日则几乎没有客户端支持压缩了。

再回头来看交互性，浏览器本身并没有执行攻击的能力，这也是作者们使用第三方插件来完成攻击的原因。最终的攻击用Java实现并利用了一个Java插件已知的漏洞。这说明浏览器支持Java

① cbcrandomiv.patch，http://src.chromium.org/viewvc/chrome/trunk/src/net/third_party/nss/patches/cbcrandomiv.patch?pathrev=97269（Chromium中的NSS 1/*n*–1补丁，2011年8月18日）。

也是完成攻击的另外一个条件。

下面是对完成攻击所需条件的总结。

(1) 攻击者必须能够在靠近受害者的位置进行中间人攻击。例如，任何Wi-Fi环境或者局域网环境都可以。很强的密码学经验和编程技巧是完成攻击的必要条件。

(2) 受害者的浏览器中必须安装Java插件。Java在那个年代是广泛存在的，因此这个要求很容易满足。

(3) 为了能通过目标网站的身份验证，受害者还必须访问其他被攻击者控制的网站。这可以使用社会工程学方法来达成。同样，攻击者可以劫持任何明文HTTP网站。因为大多数网站仍然没有加密，所以这个条件可以很容易满足。

(4) 服务器必须默认使用CBC套件并且禁用压缩。说来有趣，大量服务器满足这个条件。

综上所述，在BEAST攻击被公开的时候，尽管有诸多的限制条件，执着的攻击者还是可以相对容易地展开攻击。

今天的状况发生了很大的变化，主要是因为所有的现代浏览器（也包括Java）都已经实现了BEAST攻击的防御方法。此外，由于浏览器运行Java的不安全性所导致的Java使用的压制，也使得攻击变得更加困难。这一切都是假设你的用户群已经升级他们的软件，某些依然运行老旧软件的用户则还会受到威胁。

整个生态系统正在缓慢地支持TLS 1.2协议，虽然全部支持TLS 1.2依然还需要很长的时间。不过遭受BEAST威胁的用户和服务器的规模变得越来越小，由此造成的风险现在也变得非常低。

7.3　压缩旁路攻击

压缩旁路攻击属于消息长度旁路攻击的一种特殊形式。我们来假设一个场景：你可以观察到某个人在使用网上银行时的加密通信数据。为了获取到账户的当前余额，网上银行的客户端可能会调用一个API。仅仅通过观察响应的大小就可以评估出大致的数值：富裕受害者的账户余额会有更多的数字，这使得响应的长度更长。

结果就是当使用了压缩并且攻击者可以提交自己的数据进行压缩的时候，就形成了一个压缩预示。在本节中，我将讨论一系列与压缩相关的、针对TLS的攻击，包括CRIME、TIME和BREACH。

7.3.1　压缩预示如何生效

压缩在这个上下文中很有趣，因为它改变了数据的大小和数据本身的差异。如果你仅能观察到压缩率，那么恐怕你将无法进行有价值的攻击，因为只能推断出数据是否被进行了较好的压缩。最好的结果就是你可能会基于此来分辨出不同的数据类型。例如，文本的压缩效果比较好，而图片则不太好。

如果你可以提交自己的数据进行压缩，并且将它与其他加密数据（要解密的内容）混合起来，那么这个攻击会变得更加有趣。在这种情况下，你的数据会影响压缩的过程；通过改变数据，你可以观察还有其他哪些数据会一并被压缩。

为了了解为什么这种攻击十分严重，我们需要来看一下压缩是如何工作的。本质上，所有的无损压缩算法都是通过去掉重复的方式展开工作的。如果某个字符串重复出现了2次以上，在输出中则只会包括该字符串的一份副本，并带有该份副本会出现在哪里的位置信息。例如，以非常流行的LZ77算法为例，它的工作原理如图7-4所示。

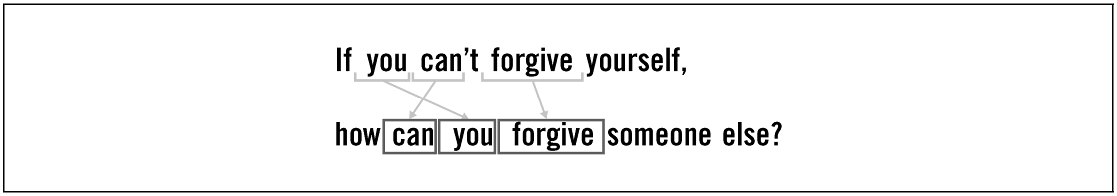


图7-4　压缩算法通过识别并去掉重复的方法来减小数据大小

我们说一个预示（Oracle）是存在的，如果你能让任意的数据（猜测数据）和某些加密数据在同一个上下文中被压缩。通过观察压缩数据的输出，你能确定猜测是否正确。如何确定？如果你猜测的是对的，那么压缩就会生效并使得压缩后的数据变小，这样你就知道你猜测的数据是正确的。如果你提交随机内容，因为没有压缩，所以输出的数据长度会变大（如图7-5所示）。

7



图7-5　压缩预示：一个正确的例子和一个不正确的例子

在接下来的几节中，你会看到虽然开展实际的攻击还存在诸多障碍，但是攻击背后的理论就是如此简单。

信息泄露是TLS协议的缺陷吗？

信息泄露看起来是SSL/TLS协议的缺陷，但是实际上这只是一个被记录在案的局限性。下面是TLS 1.2协议的相关部分：

> 任何被设计为基于TLS使用的协议都必须进行小心地设计以处理所有可能的、针对其进行的攻击。作为一种实践，这意味着协议的设计者需要了解TLS提供以及不提供哪些安全属性，并且明白不能对后者产生依赖。
>
> 从实际情况来说，TLS记录的类型和长度是不被加密保护的。如果此信息本

身是敏感的，应用层协议的设计者需要采取措施（填充、覆盖流量）来最大程度地减少信息泄露。

有些人可能会说真实的漏洞其实是浏览器允许攻击者对受害者进行高级别的控制，这可能是真的。适应性的明文攻击在明码学领域是一个大问题，但是我们有TLS，这个经常被使用在某些超出其原始设计的场景中的协议。

所有基于浏览器的攻击都基于这样一个事实，那就是攻击者可以在真实用户会话的上下文中提交请求，这造成了攻击者提供的数据连同受害者的数据一起发送。没有人会反对说这种情况很正常。如果我们能接受一个随机的页面向任意网站提交数据的话，那至少应当保证这个行为发生在独立的环境中（例如沙盒中）。

不幸的是，整个互联网是以一种混乱纠缠的方式进化发展，也就是说强制使用严格的分离策略会使得大量的网站无法使用。迟早，该方案可能会以选择性区分的方式来提供，这允许网站来声明自己的安全空间。

对于隐藏长度，即使这种方案被实现了，也会有很多关于其有效性的质疑。这种方案可能不会在所有场景下都能正常工作。某些高安全级别的网站通过使用固定长度的方法来解决此问题，例如使用底层协议提供的全部底层带宽。然而这种方案显然对于大多数网站来讲过于昂贵。

7.3.2 攻击的历史

压缩旁路攻击首先由John Kelsey发现。在2002年发表的一篇论文中[①]，他展示了一系列的攻击场景，每种场景的效果均不同。这其中就有对敏感数据的提取，并在之后被改进成了在浏览器环境中执行。2002年的时候条件与现在区别很大，并且真实的攻击很难发生。因此，John Kelsey总结道：

> 字符串提取攻击对于很多系统都不是切实会发生的，因此这种攻击要求获得部分选择明文的能力。

压缩旁路攻击在几年后的新闻中出现，虽然那时还不是针对TLS协议。2007年，一组研究人员率先开发出了在加密网络电话流量中识别出语音的算法[②]，并在那之后将语音短语的识别成功率提高到平均50%，对于某些特殊的短语可以达到90%。[③]

在之后的几年内，浏览器持续进化，导致了适应性选择明文攻击成为可能，而且有可能是针

① Compression and Information Leakage of Plaintext，http://www.iacr.org/cryptodb/data/paper.php?pubkey=3091（John Kelsey，FSE，2002年）。

② Language identification of encrypted VoIP traffic: Alejandra y Roberto or Alice and Bob，https://www.cs.jhu.edu/%7Ecwright/voip-vbr.pdf（Wright等，USENIX Security，2007年）。

③ Uncovering Spoken Phrases in Encrypted Voice over IP Conversations，http://www.cs.unc.edu/%7Efabian/papers/tissec2010.pdf（Wright等，*ACM Transactions on Information and System Security*，第13卷，编号4，文章35，2010年12月）。

对几乎所有人都有效的。在2011年，BEAST攻击展示了攻击者如何控制受害者的浏览器以执行对加密内容的混合攻击。

2011年8月，由压缩旁路攻击而引发的隐私问题在SPDY[1]的开发邮件列表中进行讨论。[2]实际上，下面这段来自Adam Langley的摘录描述了一次针对浏览器的压缩旁路攻击是如何进行的。

> 攻击者的脚本在evil.com上运行。同时，受害者的客户端打开了一个到victim.com的压缩连接，并且用户已经登录，且带有一个登录Cookie。evil.com可以向victim.com发起请求，例如在页面中加一个<img>标记并且将src指向victim.com……攻击者可以对流量进行观察并评估发送的请求的大小。通过调整URL，攻击者可以尝试最小化请求大小，例如当URL与Cookie值匹配的时候。
>
> 我尝试过使用HTTP请求来实施上述步骤，并发现非常容易就可以获取到Base64编码的Cookie值的前5字节……这是一个实际攻击并可以基于此形成一篇不错的论文，当然前提是有人有时间来写。

7.3.3 CRIME

一个实际的压缩旁路攻击发生在2012年，被命名为CRIME攻击，由Duong和Rizzo设计，他们同时也是BEAST攻击的发明者。CRIME通过使用恶意的JavaScript脚本来实现TLS压缩旁路攻击从而成功地在主动中间人攻击中提取到了客户端的Cookie。该攻击的官方披露是在2012年9月的Ekoparty大会上进行的，[3]但是由于早期非正式的新闻稿[4]中暴露了足够多的信息，安全专家可以成功地猜测到攻击的真实情况[5]。

之后，在一些投机分子的合作之下，概念验证的方案被公布出来。[6]随着攻击手段泄密，进一步的信息以及一段演示视频在会议开始前几天公开。[7]CRIME的发现者一直没有公开他们的代码，但是声称他们可以仅使用6个请求来破解Cookie中的一个字符。

实现CRIME攻击的原理与实现BEAST一样：攻击者必须能够操作受害者的浏览器来向目标服务器提交大量的请求，与此同时观察网络上发出的数据包。每次请求都将是一次猜测，就像在

---

① SPDY是一个相对比较新的协议，由Google设计用来加速Web浏览。

② Compression contexts and privacy considerations，https://groups.google.com/forum/#!msg/spdy-dev/B_ulCnBjSug/rcU-SIFtTKoJ（Adam Langley， 2011年8月11日）。

③ The CRIME attack，http://netifera.com/research/crime/CRIME_ekoparty2012.pdf（Duong和Rizzo，Ekoparty Security Conference 9° edición， 2012年）。

④ New Attack Uses SSL/TLS Information Leak to Hijack HTTPS Sessions，https://threatpost.com/new-attack-uses-ssltls-information-leak-hijack-https-sessions-090512/76973/（Threatpost，2012年9月5日）。

⑤ CRIME - How to beat the BEAST successor，http://security.stackexchange.com/questions/19911/crime-how-to-beat-the-beast-successor（Thomas Pornin，2012年9月8日）。

⑥ It’s not a crime to build a CRIME，https://gist.github.com/koto/3696912（Krzysztof Kotowicz，2012年9月11日）。

⑦ Crack in Internet’s foundation of trust allows HTTPS session hijacking，http://arstechnica.com/security/2012/09/crime-hijacks-https-sessions/（Ars Technica，2012年9月13日）。

7.3.1节中讨论的那样。与BEAST不同，CRIME对于请求的内容和时机没有较强的要求，这使得攻击的实施将更加简单并且只需要使用浏览器的本地功能。

1. TIME

在CRIME公开之后，很快就出现了其他的改进型攻击手段。2013年3月，Tal Be'ery在2013年欧洲黑帽子大会上演示了TIME攻击。[①]CRIME攻击最大的限制是攻击者必须能访问到本地网络以便对数据包进行观察。虽然TIME攻击依然使用压缩作为原理，但是它将JavaScript脚本进行了扩展，使用I/O时长的差异来判断压缩记录的大小。该方法十分直截了当，使用`<img>`标记构造从受害者浏览器发出的请求并使用`onLoad`以及`onReadyStateChange`事件来评估时间。这样，整个攻击只在浏览器中发生。

通过这种变化，攻击现在几乎能针对互联网上的任何人展开，只要你能让受害者运行特定的JavaScript即可。在实际中，这可能会要求使用一些社会工程学方法。

还有一个疑问依然存在。CRIME通过观察压缩输出中1字节的差异来实现攻击，如果使用时间，是否真的能区分到如此小的差异？事实证明，这是可行的，通过在网络层玩一个小把戏来实现。

在TCP协议中，协议做了大量的工作来避免通信的另外一段被发送大量的数据以致无法处理。这么做的原因是：同样通信的双方在地理距离上很远。例如，一个数据包从伦敦到纽约需要45毫秒的时间。如果你每次发送一个包然后等待对端的确认，则你在每90毫秒里只能发送一个包的数据。为了让通信速度更快，TCP允许通信双方每次发送多个数据包。然而，为了确保这种行为不会导致通信的某端负担过大，他们需要使用一个实现协商好的限制，这个限制叫作拥塞窗口（如图7-6所示）。拥塞窗口在开始的时候很小，之后随着时间来增长，这种方式称为慢启动。

初始拥塞窗口的大小区别很大。老的TCP协议栈可能会使用一个5 ~ 6 KB的较小窗口，但是最新的流行协议栈将此大小增加到了15 KB。TIME攻击在各种大小的窗口下都可以进行。在下面的例子中，我假设客户端使用一个大小为5 KB的初始拥塞窗口（3个数据包）。

在连接开始的时候，如果你要发出的数据的大小没有超过初始的拥塞窗口大小，数据将一次被全部发送。但是如果你的数据很大，则第一次会尽可能多地发送数据，然后等待服务器端的确认，之后再发送剩下的数据。这样会增加一次RTT（round-trip time，往返时间）。对于伦敦–纽约链路来说，这将是一个90毫秒的额外延迟。为了将此特征作为一个时间预示来使用，你需要增大数据的大小直到完全填满初始拥塞窗口。如果再增加1字节，请求就会导致一个额外的RTT，这种延迟可以从JavaScript中判断出来。那么这样你就可以开始利用压缩：如果你构造的数据导致压缩的大小减小了1字节，请求将会导致较短的RTT。再之后，攻击就与之前描述的一样了。

---

① A Perfect CRIME? TIME Will Tell，https://media.blackhat.com/eu-13/briefings/Beery/bh-eu-13-a-perfect-crime-beery-wp.pdf（Tal Be'ery和Amichai Shulman，2013年3月）。

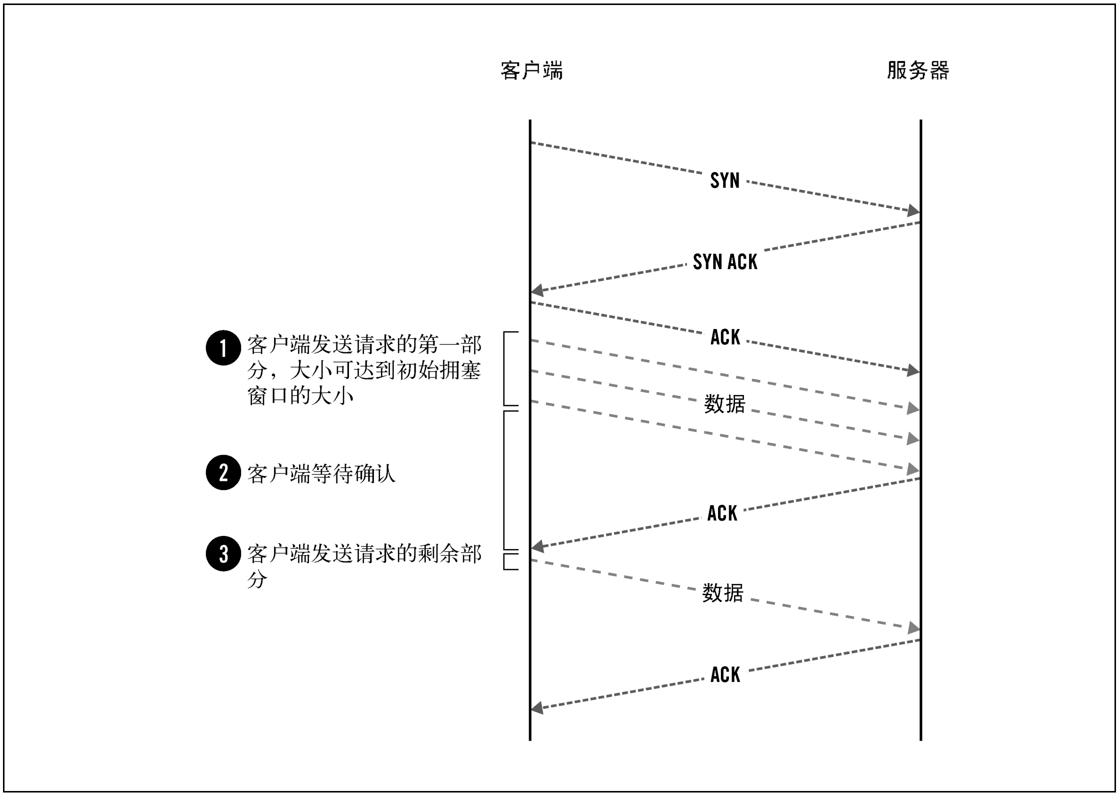


图7-6 使用TCP初始拥塞窗口大小作为时间预示

针对HTTP请求的攻击比较容易，因为你可以直接控制发送的内容。这允许你提取到浏览器中含有的机密数据，例如会话Cookie。如果你希望针对HTTP响应展开攻击，那么将会更加困难。

- 响应压缩是在服务器端执行的，这意味着你需要了解服务器的初始拥塞窗口，而不是客户端的。
- 你必须有向包含有机密数据的页面注入数据的能力。在实际中，这表示服务器应用程序得向你镜像一些你发送给它的数据。
- 在对响应进行时间评估的时候，你需要面对服务器和客户端双方可能造成的窗口溢出，这使得判断是什么导致延迟更加困难。

另外一方面，不像TLS压缩，HTTP层的响应压缩很普遍。压缩旁路攻击针对两种情况都能奏效。

众所周知，TIME攻击并没有在概念验证之外的领域广泛发生。在实际中，存在着很多阻碍实施实际攻击的障碍。例如，TIME攻击的发现者提到，由于网络抖动的原因，他们不得不重复发送多次相同的请求以便可靠地找到边界值。除此之外，拥塞窗口的大小在同一个连接中是随着时间不断增长的，这意味着你需要每次都使用一个新连接来对时间进行评估。然而，大部分服务

器会使用连接保持的功能来解决性能问题，并且你没办法通过JavaScript来控制与服务器之间的连接。因此，攻击可能会变得十分缓慢：发起一个连接，然后等待浏览器与服务器断开连接，然后不断重复这个行为。总之，即使提取16字节的数据，也需要花费很长时间。

2. BREACH

另外一个针对HTTP响应的压缩旁路攻击叫作BREACH，是在2013年8月发现的。[①]BREACH的作者希望能为大众演示CRIME攻击对于HTTP响应也同样有效。他们利用了同样的攻击手段（一个主动中间人攻击），并且开发出了真正起作用的攻击。他们最大的贡献在于对威胁的分析以及开发了实际的演示。例如，他们使用这种攻击手段攻击了Outlook Web Access（OWA），展示了他们可以获取到95%正确度的CSRF令牌，并在30秒内完成攻击。[②]

BREACH的作者将他们的成果发表在了一个网站上[③]，相关的概念验证代码位于GitHub[④]。

3. 攻击细节

BREACH从概念上就与CRIME不同，它要求攻击者可以访问受害者的网络并可以在受害者的浏览器中运行JavaScript。它们的攻击面有所不同。HTTP响应压缩只针对响应正文，这意味着响应头中的信息无法被破解。不过，响应正文中通常也存在有价值的敏感数据。作者们把精力集中在破解CSRF令牌上（方法在下面介绍），这样就可以伪造受害者的身份。

为了展开攻击，攻击者需要在响应正文中找到一个注入点。在OWA的场景下，请求中的id参数会被响应正文携带回来。因此，如果攻击者提交了如下带有攻击负载的请求：

```
GET /owa/?ae=Item&t=IPM.Note&a=New&id=INJECTED-VALUE
```

响应正文将会包含注入的值：

```
<span id=requestUrl>https://malbot.net:443/owa/forms/
basic/BasicEditMessage.aspx?ae=Item&amp;t=IPM.Note&
amp;a=New&amp;id=INJECTED-VALUE</span>
```

这已经足够用来破解响应正文中的任何加密数据了。例如，对于一个CSRF令牌来说：

```
<td nowrap id="tdErrLgf"><a href="logoff.owa?
canary=d634cda866f14c73ac135ae858c0d894">Log
Off</a></td>
```

为了建立基准，攻击者需要首先提交canary=作为攻击负载。因为内容重复，所以压缩后的响应正文会变小，这可以通过观察网络数据获知。之后就与CRIME的步骤一致了。

虽然攻击在一开始的时候看起来比较简单，但是实际中还有以下几个问题需要处理。

- 霍夫曼编码

  互联网上大部分压缩是DEFLATE压缩，实际上是两个算法的组合：LZ77和霍夫曼编码。

① BREACH: Reviving the CRIME Attack，http://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf（Gluck等，2013年8月）。

② 作者在2013年的美国黑帽子大会上演示的BREACH攻击，是在名为SSL, Gone in 30 seconds的会议上进行演示的。

③ BREACH web site，http://breachattack.com/（检索于2014年7月16日）。

④ BREACH repository，https://github.com/nealharris/BREACH（Neal Harris，检索于2014年7月16日）。

前者被我们利用来进行攻击，但是后者则使得攻击变得更加困难。霍夫曼编码是一种变长编码，基于现实中某些字符出现的频率更高的事实。正常来讲，我们使用1字节来表示一个字符。为了节省空间，我们可以将频繁出现的字符使用更短的方式表示（几位而不是1字节），然后对不经常出现的字符使用更长的表示方法。

霍夫曼编码会扰乱正确和不正确的猜测结果的长度。为了处理这个问题，必要的方法是增大请求数量为两倍，每次猜测使用两个请求。

- 分组加密

  概念攻击针对加密是行之有效的，但是只限于流加密，因为数据长度是直接反映到密文中的。当使用的是分组加密的时候，密文的大小是以整个分组进行增长的，例如在128位的AES算法中，每次增长16字节。在这种情况下，会产生数据填充以满足分组大小。因此，需要多个请求。一旦你发现了填充的大小，就可以开始进行猜测。对于每次猜测，删除填充的1字节。

- 响应内容多样性

  对于针对HTTP响应的攻击来说（TIME和BREACH），诸如文本格式、代码实践以及编码等多样性问题使得攻击更加困难。例如，攻击要求一个已知的前缀来实施攻击，但是有时候加密数据的前缀不可注入（例如引号）。抑或响应数据的长度可能发生变化，使得攻击更加困难。

CRIME攻击的作者使用了一种有趣的技术变种来实施针对TLS压缩的攻击。TLS记录大小被限制在16 KB（16 384字节），这意味着压缩可以处理的最大数据长度。之所以说这个有趣，是因为攻击者可以完全控制最开始的16 KB的数据。具体如下所示。

(1) 对于`GET`请求，最开始的5字节都是一样的：请求的方法（`GET`）、一个空格以及URL的第一个字符（`/`）。如果你在URL之后增加了16 379字节的随机数据，就可以填满整个TLS记录。你可以提交这个请求并观察压缩的大小。

(2) 你现在可以开始减小URL中的随机数据的数量，每次减小1字节。有些字节是可以预测的（例如，HTTP/1.1），但是在某个位置上你总会遇到第一个未知的字符。

(3) 现在你有了16 383字节大小的已知数据和1字节的未知数据。你应该将其作为一个请求提交。然后，在不提交其他任何请求的情况下，你需要建立一个候选字符的列表，模拟最初的16 KB数据并试着用相同的压缩算法对其进行压缩，然后将压缩之后的大小与真实请求的大小相比较。在理想的情况下，只会有一个匹配，这样就知道了未知字节。

这个方法很巧妙，因为它不需要使用太多的请求。从另外一个方面讲，攻击者使用的压缩库，需要能够对相同的输入产生相同的输出。在实际上，不同的压缩设置和不同版本的库可能会带来变化。

4. 对TLS压缩和SPDY的影响

在本节中，我将讨论一下针对TLS压缩或者SPDY进行压缩旁路攻击的一些必要前提条件。在这两种情况下，攻击都是针对请求头展开的，这就使得会话Cookie成为了最大攻击目标。

- 主动中间人攻击

  CRIME要求攻击者可以访问受害者的本地网络。因此这是一个基于本地网络的攻击，也就是说在同一个局域网或者同一个Wi-Fi环境下，攻击将会更加容易展开。攻击可以是被动的或者主动的，后者会给攻击者带来额外的灵活性。

- 客户端控制

  攻击者还需要对受害者的浏览器有足够的控制以便向目标网站发出任意请求。你可以通过使用恶意JavaScript脚本来完成这个工作，但是更加简单的方法是使用一些含有精心构造的URL的`<img>`标记。

  这可以通过社会工程学的手段达成，当然，更加普遍的方法是向受害者在被攻击的时候会访问的明文网站注入HTML代码。

- 存在威胁的协议

  就像CRIME的作者说的那样，压缩无处不在。他们详细说明了针对TLS压缩以及SPDY的攻击细节。在细节公布的时候，我正巧可以使用SSL Pulse的统计数据以及部分SSL Labs网站的统计数据来评估在服务器端和客户端的压缩支持情况。对于TLS协议，SSL Pulse的数据指出大约有42%的服务器支持压缩。虽然只有2%的服务器支持SPDY，但是这些都是一些大型网站（例如Google和Twitter）。

  这说明了，只有在两边都需要支持压缩的情况下，攻击才能发生。这说明了实际情况还没这么糟糕，因为对于浏览器厂商来说，实现TLS压缩的优先级一向不高[①]。Chrome是当时唯一支持TLS压缩的浏览器。Firefox已经实现过压缩，但是据我所知，相关代码从来没有包含到正式版本中。两家浏览器厂商都了解到了CRIME攻击，因此他们都默认关闭掉了对TLS压缩的支持。基于访问SSL Labs网站的数据，我得出的结论是大概7%的客户端支持压缩。

  作为对CRIME攻击的回应，大多数厂商对他们的产品和库进行了修复以关闭TLS压缩功能。

- 准备

  此攻击不是随意针对任何网站都可以展开的。例如，为了开始攻击，必须使用一个已知的URL前缀作为攻击的开始点。因为这些信息在不同网站之间区别很大，所以一些事先的研究是必要的，不过进行这些研究的工作量也不是特别大。

- 结论

  最好的情况是，攻击者可以获取到HTTP基本身份验证的密码。在实际中，这种身份验证方法不是很常用，这使得会话Cookie变成了更主要的攻击目标。因此如果攻击成功之后，攻击者就可以使用受害者的身份获取一切相关信息。

5. 对HTTP响应压缩的影响

对于HTTP压缩，旁路压缩攻击的影响区别很大：(1) 攻击面很大并且很难减小；(2) 完成攻击要做大量的准备，但是收效较小。

① 关注性能的网站会一直打开HTTP响应压缩，这扩大了带宽。尝试对已经压缩过的流量再次进行压缩只会无谓地消耗CPU和内存。将压缩完全移动到TLS层是可行的，但是这样会对图片进行压缩，而一般这样的压缩效果都不好。

展开针对HTTP压缩的攻击的先决条件与之前的场景一样，攻击者必须要能够对网络通信进行控制并且可以对受害者浏览器进行有限度的控制。当涉及其他因素时，会存在一些差别。

- 攻击面
  HTTP压缩同样会遭受到压缩旁路攻击的威胁（CRIME的作者们没有花太多时间研究这个，但是其他人展开过较多研究）。不像TLS压缩，HTTP压缩提供了一个巨大的攻击面并且没法简单地关闭。很多网站严重依赖于此特性以至于没办法在没有压缩的情况下运营。这里还有一个额外的要求就是攻击者需要可以向HTTP响应中注入任意数据。这一般也是可以实现的。
- 准备
  从另外一方面来说，为了施展HTTP压缩攻击，还需要很多工作。实际上，可以这么说，对目标网站有着深入的了解是必需的。会话Cookie一般不在HTTP的响应正文中出现，这意味着攻击者必须寻找其他有价值的加密信息。寻找到这种信息可能会比较难。
- 结论
  攻击的结论将会依赖于加密信息的性质。如果攻击者知道加密数据的位置，那么他就可以破解这个数据。对于大部分应用来说，攻击者感兴趣的目标将会是CSRF令牌。如果类似这样的令牌被破解，攻击者就可以使用受害者的身份在目标网站上执行任意命令。有一些网站使用会话Cookie作为CSRF令牌，这样会话就会被成功劫持。

7

7.3.4　针对 TLS 和 SPDY 攻击的缓解方法

没人再会使用TLS压缩，CRIME终结了它。在CRIME正式披露之前，用户中的一大部分（所有Chrome的用户）都支持压缩，Chrome在2012年9月时的市场份额很难精确确定，我们假设为30%。[①]归功于Chrome的自动升级功能，Chrome支持压缩的功能很快就被关闭了。

OpenSSL曾经支持压缩，因此依然可能找到旧安装以及仍支持压缩的用户代理，但是它们都不太容易遭受攻击，因为这些都不是浏览器（也就是说，不太可能进行恶意软件注入）。

但在服务器端还依然残留着对压缩的支持。在大部分情况下，仅仅对服务器打补丁就可以实现。在撰写本书的时候（2014年7月），大概10%的服务器还在支持压缩。基于Microsoft的TLS库从来没支持过压缩，并且Nginx在很久之前就禁用了对压缩的支持，那么大部分支持压缩的服务器应该都是老旧版本的Apache。

TLS层面的压缩目前来看不太可能重出江湖。就像我之前说的那样，人们并不真正需要这个功能（如果他们使用了这个功能，那基本是因为默认启用了）。即使没有压缩作为一个预示，TLS的记录长度也不是一个好的特性。目前人们正在致力于实现长度隐藏的协议扩展。[②]

① Usage share of web browsers，https://en.wikipedia.org/wiki/Usage_share_of_web_browsers（维基百科，检索于2014年2月20日）。

② Length Hiding Padding for the Transport Layer Security Protocol，https://datatracker.ietf.org/doc/draft-pironti-tls-length-hiding/（Pironti等，2013年9月）。

对于SPDY，标头压缩在Chrome和Firefox中已禁用。现在既然已经知道问题，我们可以认为此协议的后续版本不会再有问题。

7.3.5　针对HTTP压缩攻击的缓解方法

处理HTTP中的压缩旁路攻击比较困难，即使攻击不是那么容易展开。困难是双重的：(1) 你有可能无法禁用压缩；(2) 解决方法要求应用程序作出调整，开销很大。不过有一些方法依然有较好的效果。下面是这些可能性的快速概览 。

- 请求频率控制

  TIME和BREACH的作者都提到了在某些情况下由于发送了过多的请求而被发现的情况（BREACH的作者向OWA发送了数千个请求）。基于用户会话，强制一个合理的用户请求频率可能会检测出类似的攻击，在最坏的情况下，也可以将攻击拖慢很多。这种解决方法可以在Web服务器、负载均衡器或者是WAF层面实施，这意味着实现的成本不是很大。

- 长度隐藏

  一种可能的防御办法是隐藏响应的真实长度。例如，我们可以增加一个响应体过滤器来分析HTML代码并插入随机的填充字节。空格在HTML中基本都是被忽略掉的，这可以使得攻击者的攻击变得更加困难。根据BREACH作者们的说法，随机填充可以通过对请求数量的显著变大进行统计分析而被破解掉。

  这种方法的部署，在Web服务器层面非常合适，这就不需要修改后端的应用。例如，Paul Querna提议使用块编码响应变化的方式来隐藏响应长度。[①]这个方法完全不改变页面代码，但是却改变了数据长度。

- 令牌掩码

  针对CSRF的窃取可以通过混淆的方法来解决：(1) 对于令牌串中的每个字符，都准备一个随机字节；(2) 将随机字节与对应的令牌字节进行异或；(3) 将全部随机数据也包括到输出中。这个过程是可逆的，在服务器端重复进行响应异或操作，就可以得到原始的令牌字符串。这个方法适合在框架层面实施。

- 部分压缩禁用

  当我第一次思考针对HTTP响应正文的攻击时，我想到的是引用网站头将绝对不会包含目标网站的名称（如果攻击者能实现这一步，说明他已经通过XSS对网站有了充足的控制）。最开始我提议删除掉Cookie，没有Cookie就没有用户的会话信息，也就没有攻击面了。然后社区中的人提出了更好的办法：对于带有错误引用网站信息的请求，关闭响应压缩即可。[②]这样的话，代价就是对于小部分无法携带正确引用网站信息的用户，在性能上有所减退。更重要的是，这不会有任何对业务的破坏，这与删除掉Cookie的方法是不同的。

---

① breach attack， http://mail-archives.apache.org/mod_mbox/httpd-dev/201308.mbox/%3CCAMDeyhwCYYQMK+WTfvge7y20Aq EB1=kqMHgqKYhr3kBWkZyYzA@mail.gmail.com%3E（Paul Querna，2013年8月6日）。

② BREACH mitigation，https://community.qualys.com/message/20404（manu，2013年10月14日）。

7.4 Lucky 13

2013年2月，AlFardan和Paterson公布了一篇描述多种攻击细节的论文，这些攻击可以被用来破解小块的用CBC套价加密的数据。[①]他们的成果被称为Lucky 13攻击。像BEAST和CRIME攻击一样，在Web领域，小块明文数据几乎永远指的是浏览器Cookie或者HTTP的基本身份验证数据。除了HTTP，其他使用密码进行身份验证的网络协议也可能受到此威胁。

引发问题的根本原因是填充，填充经常在CBC模式中使用，但却没有被TLS的完整性检查机制所保护。这就让攻击者有机会修改填充字节并观察服务器作出的反应。如果攻击者可以成功地观察到服务器对修改填充后的反应，这就说明必要的信息已经泄露并且可以开始进行破解了。

这是我们最近几年见到的较好的TLS攻击之一。使用注入到受害者浏览器中的恶意JavaScript脚本，攻击者需要8192次请求就可以破解1字节的明文数据（例如，Cookie或者密码中的1字节）。

7.4.1 什么是填充预示

有一类攻击可以用来针对接收方展开，如果填充信息是可以修改的话。如果加密方法对密文进行身份验证的话，这就有可能发生，例如TLS中的CBC模式就不进行身份验证。攻击者不能直接修改填充数据，因为它们是经过加密的。但是因为攻击者可以对加密后的数据进行任意修改，这就让他可以对填充数据进行猜测。我们说一个预示存在，是指攻击者可以区分出哪些修改可以在解密后被当作正常的填充而哪些修改不能。

但是你如何继续进行破解呢？考虑了所有的情况后，加密本质上是使用看起来随机的数据来隐藏（掩码）明文。如果攻击者可以获取到掩码[②]，那他就可以有效地反转加密的过程并恢复出明文。

回到填充预示的话题，每次攻击者提交一次猜测密文，并产生了正确的填充字节，她便找到了掩码的1字节。这样就可以使用这一字节来推导出1字节的明文。然后，她可以重复上述行为，直到所有的密文被破解。

填充预示攻击能够成功执行依赖于：(1) 提交很多次猜测请求；(2) 存在某个可以判断猜测是否正确的方法。某些设计得不好的协议可能不会隐藏掉填充字节有错的信息。也就是说，攻击者需要能够通过观察服务器的响应来推断出猜测结果。例如，可以通过观察响应延迟的差异来判断填充字节是否正确。

如果你希望了解关于填充预示攻击的更多细节，可以参考一下某些在线教程[③]或者通过一个

① Lucky Thirteen: Breaking the TLS and DTLS Record Protocols，http://www.isg.rhul.ac.uk/tls/Lucky13.html（AlFardan和Paterson，2013年2月4日）。

② 在CBC模式下，这里指的是异或IV之后得到的中间值。——译者注

③ Automated Padding Oracle Attacks with PadBuster，http://blog.gdssecurity.com/labs/2010/9/14/automated-padding-oracle-attacks-with-padbuster.html（Brian Holyfield，2010年9月14日）。

在线模拟的页面[①]来得到更直观的认识。

填充预示攻击问题的最佳解决方法是在处理数据之前进行完整性校验。这种检查可以防止密文被修改并能够预防填充预示攻击。

7.4.2　针对 TLS 的攻击

填充预示攻击（针对TLS和其他协议）最早由Serge Vaudenay在2001年提出（正式发布是在2002年）[②]。TLS 1.0使用`decryption_failed`警告来指明填充错误并使用`bad_record_mac`来指示MAC错误。这种设计虽然不安全，但是的确没有导致在现实生活中发生攻击，因为TLS的警告是加密的，所以网络攻击者无法区分这两种类型的错误。

2003年，Canvel等对攻击进行了改进。[③]他们使用了一种基于时间的填充预测，并演示了一次成功的针对OpenSSL的攻击。他们利用了当填充字节不正确时OpenSSL会跳过MAC计算并使响应变快的特点。作者们的概念验证攻击是针对一个IMAP服务器的，在距离攻击目标很近的情况下，他们可以在一小时内获取IMAP的密码。

填充预示之所以起作用，是基于重复猜测并确定哪些字节组合可以被解密成正确的填充字节来实现的。攻击者首先截获一些密文，将其修改后提交到服务器。大部分的猜测都将是错误的。在TLS中，每次猜测失败都会导致TLS会话断开，这意味着之前的加密数据无法继续使用。在下一次猜测的时候，攻击者需要截获另外一个合法的加密块。这也是Canvel等人攻击了IMAP的原因，因为IMAP是一个自动化服务，可以在出错之后自动重试，这对于展开攻击非常理想。

为了提高CBC模式的安全性，OpenSSL（和其他TLS实现）修改了它们的代码以减少信息泄露。[④]TLS 1.1废弃了`decryption_failed`警告并增加了以下说明。

> Canvel等实现了一种针对CBC填充字节的时间攻击，这种攻击依赖于计算MAC的时间。为了防御这种类型的攻击，实现必须确保记录处理时间是基本上相同的，无论填充字节正确与否。总体来说，最好的解决方法是即使在填充字节不正确的情况下也计算MAC然后拒绝掉这个包。例如，如果填充字节不正确，实现可以假设一个长度为0的填充然后计算MAC。**这也会留下一个小的时间通路，因为计算MAC的时间某种程度上与数据的长度有关，但是这种差异一般认为不会足够大到被利用来进行攻击，这是因为与MAC数据的大小相比，时间差异太小了。**

2013年2月，AlFardan和Paterson演示了残留的旁路空隙，实际上也是能用来进行攻击的，这是

① Padding oracle attack simulation，http://erlend.oftedal.no/blog/poet/（Erlend Oftedal，检索于2014年2月28日）。

② Security Flaws Induced by CBC Padding-Applications to SSL, IPSEC, WTL...，https://www.iacr.org/archive/eurocrypt2002/23320530/cbc02_e02d.pdf（Serge Vaudenay，第534~546页，EUROCRYPT 2002）。

③ Password Interception in a SSL/TLS Channel，http://www.iacr.org/cryptodb/archive/2003/CRYPTO/1069/1069.pdf（Canvel等，CRYPTO 2003）。

④ Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures，https://www.openssl.org/%7Ebodo/tls-cbc.txt（Moeller等，最后更新于2004年5月20日）。

使用了新的技术来发现Vaudenay的填充预示。他们将其命名为Lucky 13并且展示出CBC（就像在TLS和DTLS中实现的那样）过于脆弱以致于在很久之前就应当被废弃掉。他们同时说明了一个道理，一些小的问题，如果没有引起足够的重视，其危害随着技术的不断进化，会变得更加巨大。

7.4.3 影响

为了能够进行填充预示攻击，攻击者必须能够发起一次主动攻击，也就是说他必须能够截获并修改加密流量。除此之外，因为时间的差异非常小，所以攻击者必须距离目标服务器足够近才能检测到这种差异。攻击的作者们是在同一个局域网内进行攻击试验的。远程攻击看起来对TLS并不可行，但是对于DTLS，如果使用AlFardan和Paterson在2012年开发的时间放大技术的话，就可以实现攻击。[①]

- 针对自动化系统的攻击

  经典的全明文破解填充预示攻击都是针对自动化系统的，这些系统大多是经常与服务器交互并含有出错重试的机制。因为这种攻击会持续发出很多连接，它只针对那些将敏感信息放在同一位置的协议有效果。IMAP是一个不错的选择。此攻击需要使用840万个连接来破解16字节的数据。因为每次错误的猜测都会导致一次TLS错误，而且因为TLS被设计成在这种情况下销毁会话，会强制每次新连接与服务器进行一次全新的握手。因此，攻击比较缓慢。但是仍然在某些情况下攻击者可能有几个月的时间来完成攻击，并且可以让这个自动化过程以更快的速度发起连接。

- 在已知某些明文的情况下展开攻击

  这是一种部分明文破解的攻击，如果一个加密块中最后2字节中的一个是已知的，则该攻击可以被执行。攻击的后果就是可以在65 536次尝试后破解剩余的字节。

- 使用恶意JavaScript脚本对浏览器进行攻击

  AlFardan和Paterson的最佳攻击方法，是在受害者的浏览器中使用恶意JavaScript，瞄准的是HTTP Cookie。因为恶意脚本可以影响Cookie在请求中的位置，因此可能将加密的数据重新进行安排以至于只有Cookie中的1字节是不知道的。因为Cookie中使用的字符范围的限制，研究人员估计只要8192次请求就足以解密出1字节的明文。这种攻击最好的地方在于，所有的请求都是恶意脚本发出并处理的，所以所有的连接失败对于受害者来说都是无法感知的。此外，也不需要额外的插件或跨域权限。

7.4.4 缓解方法

AlFardan和Paterson通过多种实现证明了攻击，并将这些问题汇报给了相应的开发人员，然后再对公众披露攻击的细节。这样一来，在公开攻击细节的时候，所有的库都已经提前修正了这个问题。因此，给你的SSL库打补丁就已经足够缓解攻击了，至少一开始的时候这是有效的。

① Plaintext-Recovery Attacks Against Datagram TLS，http://www.isg.rhul.ac.uk/%7Ekp/dtls.pdf（AlFardan和Paterson，NDSS，2012年2月）。

基于TLS中CBC实现的脆弱性，如果能避免使用CBC模式的算法是最好的，但是说起来容易，做起来难。在很多场景下没有足够安全的替代品。流加密算法不使用填充，所以它们并不受此问题影响，但是TLS中唯一的流加密算法是RC4，这个算法存在其他的问题（下个小节将讨论这个问题）因此无法使用。其他的流加密算法会逐步地添加到TLS中，但这需要时间。[①]这种情况让我们只能选择已验证的GCM模式的算法，这要求TLS 1.2版本的协议支持。2014年9月，TLS协议的一个扩展改变了CBC的工作模式，使其对密文而不是对明文进行身份验证[②]，但是我们还需要继续观察这个扩展是否得到足够广泛的接受以便真正能起到缓解攻击的作用。

7.5　RC4 缺陷

RC4由Ron Rivest在1987年设计，是目前仍在使用的最古老的算法之一，不考虑其存在的多种缺陷，RC4时至今日仍然非常流行。RC4之所以非常流行，既与其出现时间较早有关，也因为其实现起来非常简单并且性能较高。

今天，我们都知道RC4是可被破解的，但是针对RC4的攻击还没有进化到足够在实际场景中展开。基于这个原因，以及在很多环境中RC4的替代者存在更多的问题，因此RC4仍然在被使用着（当然，更大的原因是人们的惰性以及很多人都不知道RC4存在问题需要禁用）。

如果可能的话，应该完全避免使用RC4。例如，TLS 1.2协议提供了安全的替代算法，这说明RC4不应该被使用。然而在实际中，你可能有一些比较好的理由继续来使用它，就像我接下来要讨论的那样。

7.5.1　密钥调度弱点

在很长一段时间内，RC4已知的最大问题是其密钥调度算法的缺陷，该问题由Fluhrer、Martin和Shamir在2001年公布。[③]他们发现大量的密钥都存在一个弱点，就是密钥中的一小部分可以决定初始化输出的大部分内容。在实际中，这意味着如果密钥的一部分被使用了一段时间，攻击者就可以：(1) 破解密钥流的一部分（例如利用某些位置上的已知明文）；(2) 破解其他所有流上这些位置上的明文。此发现主要是针对WEP协议进行攻击[④]。最初实现的针对WEP的攻击，需要10 000 000条消息来破解密钥。后来该攻击被改进为只需要100 000条消息即可完成。

TLS不受此问题影响，因为每个连接都会使用新的密钥。因此RC4仍然被广泛使用，因为这

① ChaCha20 and Poly1305 based Cipher Suites for TLS，https://datatracker.ietf.org/doc/draft-agl-tls-chacha20poly1305/（Langley和Chang，2013年11月）。

② RFC 7366: Encrypt-then-MAC for TLS and DTLS，https://tools.ietf.org/html/rfc7366（Peter Gutmann，2014年9月）。

③ Weaknesses in the Key Scheduling Algorithm of RC4，http://www.crypto.com/papers/others/rc4_ksaproc.pdf（Fluhrer、Mantin和Shamir，2001年）。

④ WEP不经常重用密钥，但是它会从一个主密钥中导出新的密钥，这是通过使用一种连接的方式来完成的，这样就导致会话密钥和主密钥很相似。举例来说，因为TLS使用了散列，所以无法从连接的密钥中反推出主密钥。

个已知的问题并不影响到其在TLS中的使用。[①]尽管存在一些已知问题，但是RC4依然是TLS中使用最广泛的加密算法。我在2010年所做的关于SSL使用情况的大规模问卷调查结果显示，RC4是最受欢迎的加密算法[②]，有98%的服务器支持使用RC4算法[③]。了解密钥调度算法弱点的人们不喜欢RC4因为其有可能被误用，他们不会在新的系统中继续使用RC4。[④]

当BEAST攻击早2011年被确认之后，使得所有的分组加密算法变得不安全（即使BEAST只在TLS 1.0以及更早的协议中有效，但是在当时TLS 1.1协议还没有诞生）。因为RC4是流加密算法，对BEAST免疫，所以就成了TLS协议中唯一安全的加密算法。2013年3月，当全新的、毁灭性的漏洞在RC4中被发现的时候，ICSI证书公证人项目显示，互联网上有约50%的流量在使用RC4。在写这本书的时候，也就是2014年7月，RC4的比例是26%。[⑤]

7.5.2 单字节偏差

加密偏差（encryption bias）问题是密码学专家广泛担忧RC4的另外一个原因。早在2001年，人们发现密钥流中的某些字节出现的频率要比其他字节高。[⑥]具体来说，密钥流的第二字节更倾向于为0的概率是1/128（比正常的1/256高了2倍）。

为了理解偏差如何可以导致破解明文内容，我们需要回过头来先看一下RC4的工作原理。RC4是一种流加密算法：在初始化阶段完成之后，该算法产生一个无穷的流数据。这个数据应当是完全随机的，然后此数据被用于与明文内容进行异或，一次一字节。这个异或操作会让明文被完全掩码成随机数据，除非在拥有RC4密钥的情况下，否则无法破解。

当我们说到偏差，指的是某些值出现的概率比其他的要大。最坏的情况就是刚才讲到的倾向于0的这种。为什么？因为1字节与0进行异或，结果还是这1字节。因此，只要我们知道RC4密钥流的第二字节倾向于为0，那么我们就知道经过加密的密文的第二字节其实与明文是相同的！

为了展开这种攻击，你需要获取到相同明文被多种不同密钥加密的密文。对于TLS来说，这就要针对多个连接展开攻击。[⑦]然后你来观察第二字节，如果这个位置的值重复出现得比较频繁，

① RSA Security Response to Weaknesses in Key Scheduling Algorithm of RC4，http://www.emc.com/emc-plus/rsa-labs/historical/rsa-security-response-weaknesses-algorithm-rc4.htm（RSA Laboratories Technical Note，2001年9月1日）。

② 说来有趣，只有大概一半的TLS服务器强制了套件偏好。另外一半则使用浏览器提交来的套件列表中的第一个。

③ Internet SSL Survey 2010 is here，http://blog.ivanristic.com/2010/07/internet-ssl-survey-2010-is-here.html（Ivan Ristić，2010年7月29日）。

④ What's the deal with RC4，http://blog.cryptographyengineering.com/2011/12/whats-deal-with-rc4.html（Matthew Green，2011年12月15日）。

⑤ The ICSI Certificate Notary，http://notary.icsi.berkeley.edu/#connection-cipher-details（International Computer Science Institute，检索于2014年7月16日）。

⑥ A Practical Attack on Broadcast RC4，http://saluc.engr.uconn.edu/refs/stream_cipher/mantin01attackRC4.pdf（Mantin和Shamir，2011年）。

⑦ 在密码学中，这个被称为*多会话*（multisession）攻击。这个名称在TLS的上下文中可能会比较混淆，因为一个TLS *会话*（TLS session）是一组密码学参数，这组参数通过会话复用机制可以被多个连接使用。即使在会话复用的情况下，TLS也为每个连接生成新的对称密钥。

则这个值极有可能就是明文内容的相同值。可能会需要一些猜测，但是你能观察到的密文越多，你猜测成功的概率就越大（如图7-7所示）。

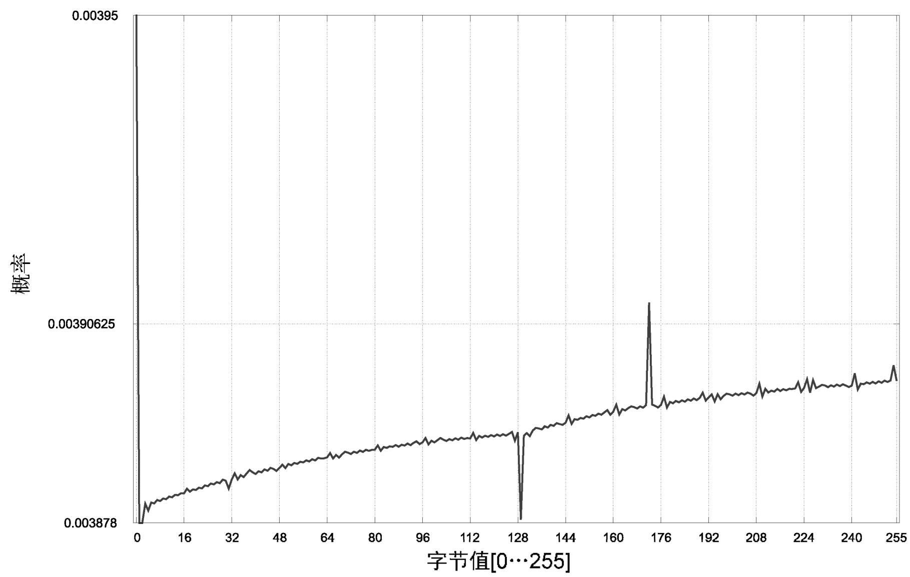


图7-7　RC4密钥流第二字节偏差（来源：AlFardan等，2013）

我们通过这些偏差能获得什么，取决于协议上的设计。第一个要素是，有效的数据确实被放在了制定的位置上。例如，在TLS协议中，最开始的36字节一般被Finished协议消息使用，并且在每次连接的时候都发生变化，因此没有长效数据。[①]对于TLS协议，第二字节偏差没什么用途。

第二个要素是，可以在大量的连接上，确保每次获取的数据都是相同的，并且每次都出现在同一个位置。对于某些协议来说，这不是问题。例如在HTTP中，Cookie和密码在每个请求中的位置都相同。

7.5.3　前 256 字节偏差

2013年3月，AlFardan等人发表了一篇论文，该论文描述了新发现的RC4漏洞以及两种针对TLS的有效攻击。[②]

① 某些协议扩展增加了也进行了加密的其他消息。例如，用于协商SPDY的次协议协商（next protocol negotiation，NPN）扩展就是如此。不像Finished消息那样，内容是完全随机的，因此其他消息可能会被利用RC4偏差来实现攻击。

② On the Security of RC4 in TLS and WPA，http://www.isg.rhul.ac.uk/tls/（AlFardan等，2013年3月13日）。

其中一个攻击是基于RC4偏差并不是局限于某些位置的某些字节。通过生成并分析2^{44}个RC4密钥的密钥流，AlFardan等人发现在最开始的256字节中均存在着多种偏差。他们进一步改进了算法来处理独立位置上的多种偏差（例如某字节出现10或者23的概率较大，但是出现其他值的概率却相同）。最后的结论是需要2^{32}个数据样本来破解全部的256字节的内容，成功率几乎是100%。在被攻击数据的字符集合有限的情况下（例如密码和HTTP Cookie），数据样本的数量可以减少到2^{28}。这与RC4承诺的2^{128}位的安全性相差甚远。

> **注意**
> 为什么在存在如此多危险迹象的情况下，在很长时间内这种偏差问题没有被及时发现？我听过的一种说法是，绝大多数密码学专家认为RC4已经被证明为不安全的，因此不需要做进一步的研究。事实上，很多密码学专业非常惊讶于RC4被使用的广泛程度。这可能是因为在TLS协议中没有针对RC4的广泛攻击导致了RC4被继续使用下去。

尽管这种攻击较为严重，但是以下诸多限制使其依然主要停留在理论阶段。

- 连接数量

  在最好的情况下，攻击者需要2^{28}个加密数据的样本。从另一个角度来看，这需要268 435 456个连接。显然，获取全部数据样本需要很长的时间并且可能使用大量的网络带宽。在可控的理想环境下，也就是两端被配置成尽可能多的生成和处理RC4连接，在启用会话恢复的情况下，研究人员们在每秒发起500个连接，总共使用了2^{25}个连接的情况下，完成了这个耗时16小时的实验。

  在与现实生活更加贴近的场景下，一个纯粹的被动攻击会耗费更长的时间。例如，假设每秒一个连接（每天86 400个连接），这需要8年时间才能收集齐全部的数据样本。

  连接速率可以通过控制受害者的浏览器来提高，强迫浏览器并行发起多个连接。这是与BEAST类似的方法。在这种场景下，需要做额外的工作来绕过连接复用并且避免多个请求经由同一个连接发送（攻击只能获取到每个连接的前256字节）。为了完成上述步骤，可以在观察到结果之后马上在TCP层使用中间人攻击的方式重置当前连接。因为TLS协议会在遇到错误的时候丢弃会话，所以在这种方式下，每次都是全握手。这将使得攻击变得非常缓慢。①

- 位置

  这是一种中间人攻击。正像前文所述，被动攻击由于无法在合理的时间内完成，因此实际上很难发生。进行主动攻击的话则需要有注入JavaScript恶意脚本和进行中间人攻击的能力。

- 范围

  这个攻击只对明文的前256字节有效。因为需要大量的数据样本来进行分析，因此很难确保在所有的样本中存在的都是完全相同的敏感数据。这个限制使得此攻击只能针对某些

① 理论上是这样。实际中，应用程序对于非正常关闭的连接是倾向于尽量容忍的，这样就容易受到截断攻击。可以在6.7节中找到更多信息。

基于密码进行身份验证的协议展开，比如HTTP、Cookie。事实证明，即使在HTTP中，攻击也不是很容易发生，因为主流的浏览器都将Cookie放在了220字节界限之后（前36字节是TLS使用的，数据价值不大）。HTTP基本身份验证在Chrome中会受到威胁，因为Chrome将密码放在了100字节左右的位置。所有其他的浏览器都将密码放置在了攻击能够破解的范围之外。

7.5.4　双字节偏差

除了单字节偏差外，RC4还存在影响到连续字节的偏差。这些偏差不出现在单一位置而是以通常的间隔连续出现在加密流中。[①]

在AlFardan等人的第二个攻击中，他们展示了如何使用双字节偏差来破解明文。利用双字节偏差的攻击的一个优点是不需要使用大量不同RC4密钥加密的数据样本。这使得攻击的效率更高，因为多个样本可以从同一个连接上获取。另外一方面，因为仍然需要不断地提交加密后的数据，因此攻击者需要几乎完全控制受害者的网络。被动攻击不可行。

双字节偏差攻击可以基于13×2^{30}个数据样本破解出16字节的明文数据。为了收集到一个数据样本，需要使用一个512字节的POST请求。假设响应的数据较小，则攻击需要消耗3.25 TB的流量。在可控的理想状态下，按照每小时收集600万个样本，这些数据样本的全部收集会花费2000个小时（83天）。

虽然这个攻击比第一种要更加实际，但是依然在现实中很难实施。

7.5.5　针对密码进行攻击的改进

2015年3月，Garman等研究人员发布了他们针对RC4的改进型攻击的细节，他们的攻击是专门针对被加密数据是密码的情况。[②]根据他们的报告，他们的攻击可以做到在使用2^{26}次加密后就可以获得不错的破解成功率，而之前的类似攻击方法需要2^{34}次加密才能破解出HTTP的会话Cookie。

同样在2015年3月，Imperva强调了RC4的*恒定弱点*（invariance weakness），该问题在2001年就发布过。[③]该研究集中在RC4是已知会偶尔生成不好密钥的问题上，该问题可能会导致TLS连接的前64字节数据被破解。[④]虽然攻击者无法影响RC4密钥的生成，但是一个被动的攻击者经过观察几百万个使用RC4加密的TLS连接，就可以在少部分场景下获取到明文。通过这种方式，大概

① Statistical Analysis of the Alleged RC4 Keystream Generator，http://www.mindspring.com/%7Edmcgrew/rc4-03.pdf（Fluhrer和McGrew，2001年）。

② Attacks Only Get Better: Password Recovery Attacks Against RC4 in TLS，http://www.isg.rhul.ac.uk/tls/RC4mustdie.html（Garman等，2015年3月）。

③ Attacking SSL when using RC4，http://www.imperva.com/docs/HII_Attacking_SSL_when_using_RC4.pdf（Imperva，2015年3月26日）。

④ 理论上讲，可能有100字节会被破解，但是TLS连接本身会使用掉前36字节，因此应用层数据最多有64字节可能被破解。

每2^{24}个连接中就有一个会被破解。

2015年7月，Vanhoef和Piessens进一步改进了RC4攻击，他们将破解一个长度为16字节的Cookie的时间减小到只有大约75小时（虽然他们声明有一个案例只需要52小时）[①]。他们的方法是执行一个BEAST风格的攻击，在该攻击中将恶意JavaScript注入到受害者的浏览器来加速攻击。然而，即使这样，依然要求受害者发出大约 9×2^{27}次请求，每个请求都含有相同密钥加密后的密文。为了在75小时之内完成攻击，研究者们需要每秒发出约4450个请求。这样大量的请求和如此快的速度让人怀疑该攻击方法是否可以在现实生活中使用。

7.5.6 缓解方法：RC4 与 BEAST、Lucky 13 和 POODLE 的比较

针对RC4的攻击很严重，因为在理想环境下可能导致明文被破解，但是在现实生活中则很难施展攻击。考虑到RC4的安全宽限期已变得非常小，最好的方法还是尽可能快地停止使用RC4。

在没有合适的备选算法的时候，停止使用RC4并不是最佳选择。这里有以下两方面的问题需要考虑。

- 互操作性

  RC4是存在很久并广泛使用的加密算法，被“承诺”为将永远存在。这样一来，可能会有一些客户端除了RC4之外不支持其他算法。然而，机会在于这种类型的客户端的数量非常小。[②]如果你有一个真正多元化的客户端群，并且你认为这些只支持RC4的客户端会导致巨大的问题发生，那就可以考虑继续保留RC4，但是要把RC4放在算法列表的最后面。因为大多数客户端会选择使用其他的算法，这样做既降低了发生攻击的可能性，也没有破坏互操作性。

- 安全性

  如果你禁用了RC4，那么就需要考虑在TLS 1.0和更早的版本中使用CBC模式算法的问题。在这种情况下，BEAST和POODLE攻击可能会发生。首先，你的服务器可能还是最高只支持TLS 1.0协议（如果是这样，先不要考虑RC4的问题，而是尽快升级到TLS 1.2）。你也许当下还没有禁用SSL 3。即使你的服务器进行了及时的更新和升级，你的客户端方面也有可能存在问题。某些客户端会容易遭到BEAST攻击。

  没有足够的数据表明哪种攻击更容易发生（BEAST、POODLE还是RC4）。它们都很难进行实施。针对RC4的攻击有可能在任何版本的协议下实施，但是这需要浏览器被注入并且有足够的时间和带宽。目前没有已知的针对POODLE的防御，虽然针对SSL 3进行攻击也需要一些额外的工作。BEAST也很难进行实施，但是如果各种条件都具备，攻击将会进行得很快。实施BEAST攻击的最大障碍是主流的平台都已经做了针对性的升级和更新，客户端方面存在漏洞的比例也一直在下降。真正的问题是：是否存在一些我们不知道的，

① RC4 No More，https://www.rc4nomore.com/（Vanhoef和Piessens，检索于2015年7月25日）。

② The Web is World-Wide, or who still needs RC4，https://blog.cloudflare.com/the-web-is-world-wide-or-who-still-needs-rc4/（John Graham-Cumming，2014年5月19日）。

> 但是却比这些攻击更好的手段？很多人都在问这个问题，特别是针对RC4这个已经被FIPS排除在外的算法。这些漏洞和弱点是否早已被NSA所了解？还有什么问题是他们早就知道了的？
>
> Lucky 13也值得担忧。虽然进行了紧急的修复，但是TLS中的CBC模式天生就不安全。乐观地看，使用TLS 1.2的客户端和服务器趋向于使用已验证的GCM模式套件，GCM不使用RC4或者CBC，因此这是目前来看解决各种TLS密码套件漏洞的最佳方法。

我们不能基于投机和偏执的心理来作决定；此外严格地说，也不存在完全正确的决定。缓解BEAST和POODLE攻击在某些场合下是合理的，禁用RC4在另外的场景下可能也是最佳选择。在这种情况下，去看看别人都怎么做，终归是有所帮助的：在撰写本书的时候，Google仍然启用了RC4，但是只针对那些不支持现代加密算法的客户端才使用（TLS 1.0和更早的版本）。

另一方面，Microsoft在Windows 8.1中大胆地禁用了RC4，以及在部分Windows 7中也是如此。Schannel在客户端模式下依然会使用RC4，但是仅当服务器端不提供其他加密算法时才会使用。Firefox也在2015年1月开始使用同样的策略[①]，虽然它现在倾向于只对在自己白名单上的服务器才使用RC4，并将此作为实现全面废弃RC4的第一步。（很难及时发现浏览器在某个时刻的行为是如何的，因为它们在每个版本中都会有变化。）

有些观点认为这种回退是必要的，因为仍然存在只支持RC4的服务器。根据SSL Pulse的数据，截至2015年7月，还有932个这样的服务器（0.6%）。

2015年2月，RFC 7465发布，明确禁止在TLS中使用RC4算法。[②]

7.6　三次握手攻击

2009年，当发现TLS的重新协商机制不安全的时候，就发明出了安全重新协商的方法来对协议进行修正（如果你还不了解这些，请阅读7.1节）；但是这个做法并不是特别有效。2014年，一组研究人员展示了他们的三次握手攻击，该攻击利用了两个独立的TLS漏洞，对重新协商再次展开了攻击。[③]

7.6.1　攻击

为了理解攻击是如何实施的，你首先需要了解安全重新协商的原理。当重新协商发生的时候，服务器校验客户端提供的前次握手中的`verify_data`值（在前次握手中的已加密`Finished`消息中存放）。因为只有客户端能知道这个值，所以服务器会确认发起重新协商的客户端还是同一个。

这看起来攻击者无论如何也不可能获取到这个值，因为在这个值传输的时候都是加密的。但

---

① Bug 1088915 - Stop offering RC4 in the first handshakes，https://bugzilla.mozilla.org/show_bug.cgi?id=1088915（Bugzilla@Mozilla，检索于2015年3月22日）。

② RFC 7465: Prohibiting RC4 Cipher Suites，https://tools.ietf.org/html/rfc7465（Andrei Popov，2015年2月）。

③ Triple Handshakes Considered Harmful: Breaking and Fixing Authentication over TLS，https://www.secure-resumption.com（Bhargavan，2014年3月）。

是现在有办法来破解出这个值并针对重新协商展开攻击。具体的攻击方法有如下三个步骤，并利用了TLS协议的两个漏洞。

1. 第一步：未知共享密钥缺陷

第一个被利用的缺陷发生在RSA的密钥交换过程中。作为TLS会话安全的基石，主密钥的生成，主要是由客户端驱动的。

(1) 客户端生成预主密钥和一个随机数并将它们发送给服务器。

(2) 服务器生成自己的随机数然后发送给客户端。

(3) 客户端和服务器从这三个值中生成主密钥。

两个随机值都是明文发送的，但是为了防止针对TLS的中间人攻击，预主密钥是加密传输的，客户端用服务器的公钥加密，这样攻击者就无法破解，除非攻击者能获取到服务器的私钥，这是攻击者的第一道难关。

三次握手攻击依赖于一个恶意网站的配合。在这种情况下，你要让受害者访问这个看起来无害的，但却在你控制之下的网站（通常的方式是使用社会工程学方法）。在这个网站上，你有你自己的有效证书。

接下来就是有趣的步骤。客户端生成了一个预主密钥和一个随机值，并发送给了恶意服务器。[①]预主密钥是加密的，但是恶意网站是这个值的目标接收方，因此可以解密开。在与客户端的握手完成之前，恶意网站向目标网站发起一个连接，并且复用了客户端发来的预主密钥和随机值。之后恶意网站将目标服务器的随机数传给客户端。当密钥交换结束后，将会形成两个TLS连接，涉及三方的通信局面。这三方共享相同的连接参数，因此拥有相同的主密钥，如图7-8所示。

这个缺陷叫作*未知密钥共享*（unknown key-share）[②]，可以明显发现这不是TLS期望的行为。然而，仅仅依靠这个缺陷，还无法展开攻击。恶意服务器现在还无法真正完成有害的行为。因为它有主密钥，所有攻击者可以看到所有的通信内容，但是这在不引入额外服务器的情况下也可以达成。如果攻击者想要达成这个效果，他可以进行网络钓鱼，钓鱼是一个严重的问题，但是那不是TLS能解决的。

> **注意**
>
> RSA密钥交换几乎到处被使用，但是同样有针对DHE密钥交换的攻击。研究者发现主流的TLS实现会接受不是素数的不安全DH参数。在TLS协议中，是由服务器来选择DH参数的。因此，恶意的中间服务器精心选择参数之后可以轻而易举地破坏DHE密钥交换。ECDHE密钥交换是DHE的一个椭圆曲线变体，无法攻破，因为没有任何TLS

① 因为恶意服务器在客户端和服务器中间，它总是可以强迫通信双方使用基于RSA密钥交换的某种套件，只要被攻击的通信双方支持这类套件，那这种强迫就可以持续进行下去。在TLS中，是由服务器来选择密码套件。当客户端向服务器发起一次握手的时候，恶意服务器只提供使用RSA密钥交换算法的套件。

② Unknown key-share attacks on the station-to-station (STS) protocol，https://books.google.co.uk/books?id=cyHE4f0ukbYC&pg=PA154（S. Blake-Wilson和A. Menezes，第154~170页，Public Key Cryptography，1999年）。

> 的实现支持任意的DH参数（使用DHE时也同样如此）。ECDHE则依赖于已命名曲线，这些曲线是公认较好的参数集。

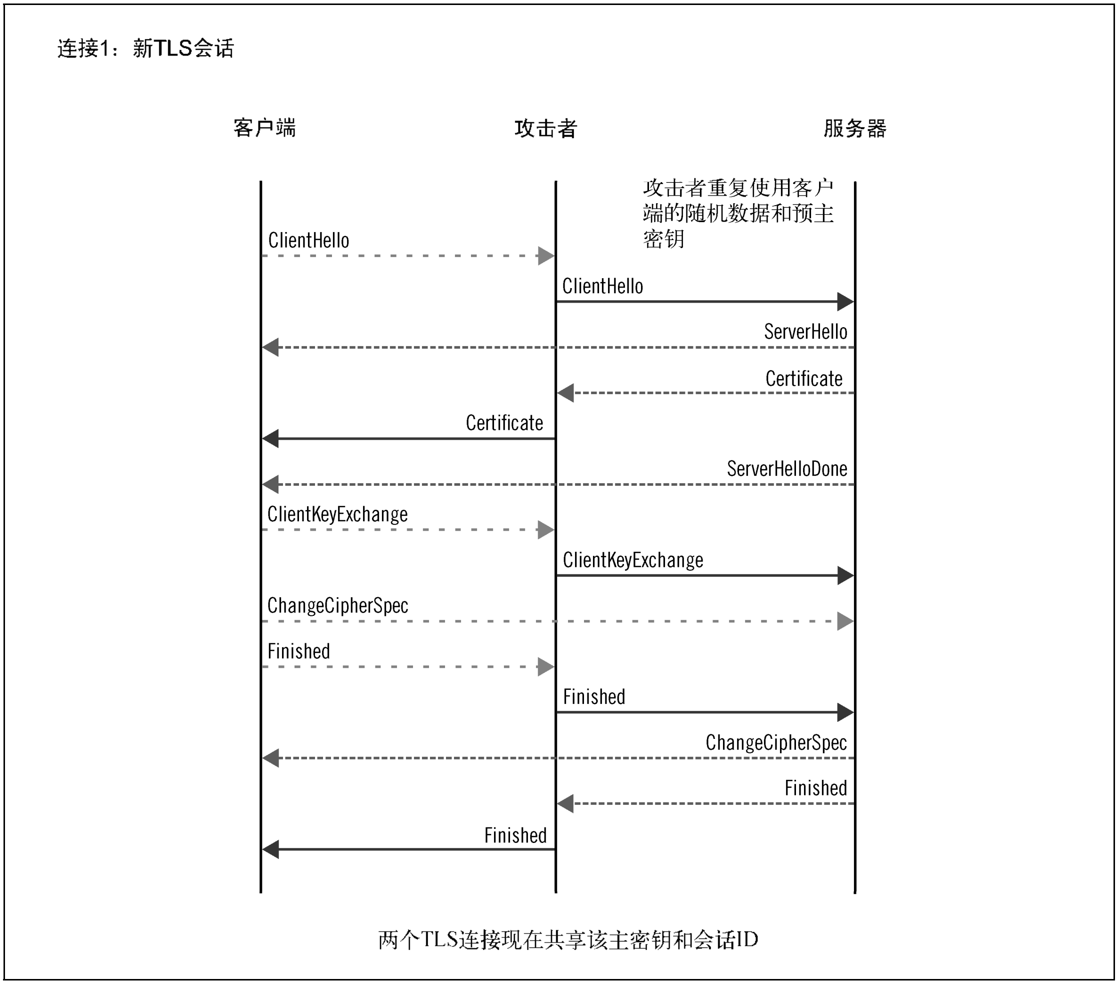


图7-8　三次握手：未知密钥共享

2. 第二步：全同步

攻击者目前无法针对重新协商进行攻击是因为每个连接都含有不同的`verify_data`值。为什么？因为证书不一样：第一个连接是恶意网站域名的证书，而第二个连接是目标网站的证书。

对于第一个连接，攻击者无能为力，但接下来，攻击者可以利用会话恢复机制并利用后续的短握手。当会话恢复的时候，是没有进行身份验证的，会话恢复仅使用主密钥就可以实现对通信双方的身份验证。

然而，当会话恢复的时候不再要求证书。因此当握手结束后，Finished消息在两个连接上将是相同的（如图7-9所示）!

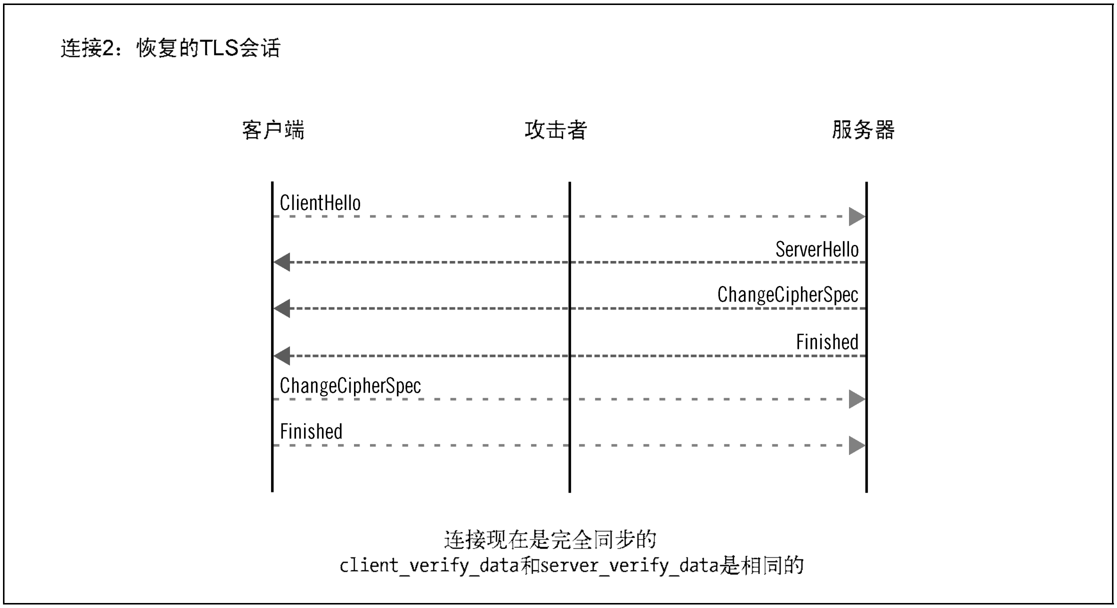


图7-9　三次握手：全TLS连接同步

3. 第三步：仿冒身份

现在攻击者可以开始发起重新协商，并利用客户端的证书伪造身份。攻击者完全控制了两边的连接，并可以发送任意数据。在目标网站这边，攻击者浏览到一个需要进行身份验证的资源，对此目标服务器发出重新协商请求并要求客户端提供证书。因为安全连接参数在两个连接上均相同，攻击者就可以复制消息，让受害者和目标服务器进行重新协商，不同的是，这次重新协商客户端会发送证书进行身份验证。至此攻击完成（如图7-10所示）。

重新协商之后恶意服务器无法观察明文流量，该服务器将继续在受害者和目标网站之间传递数据直到任意一端关闭连接。

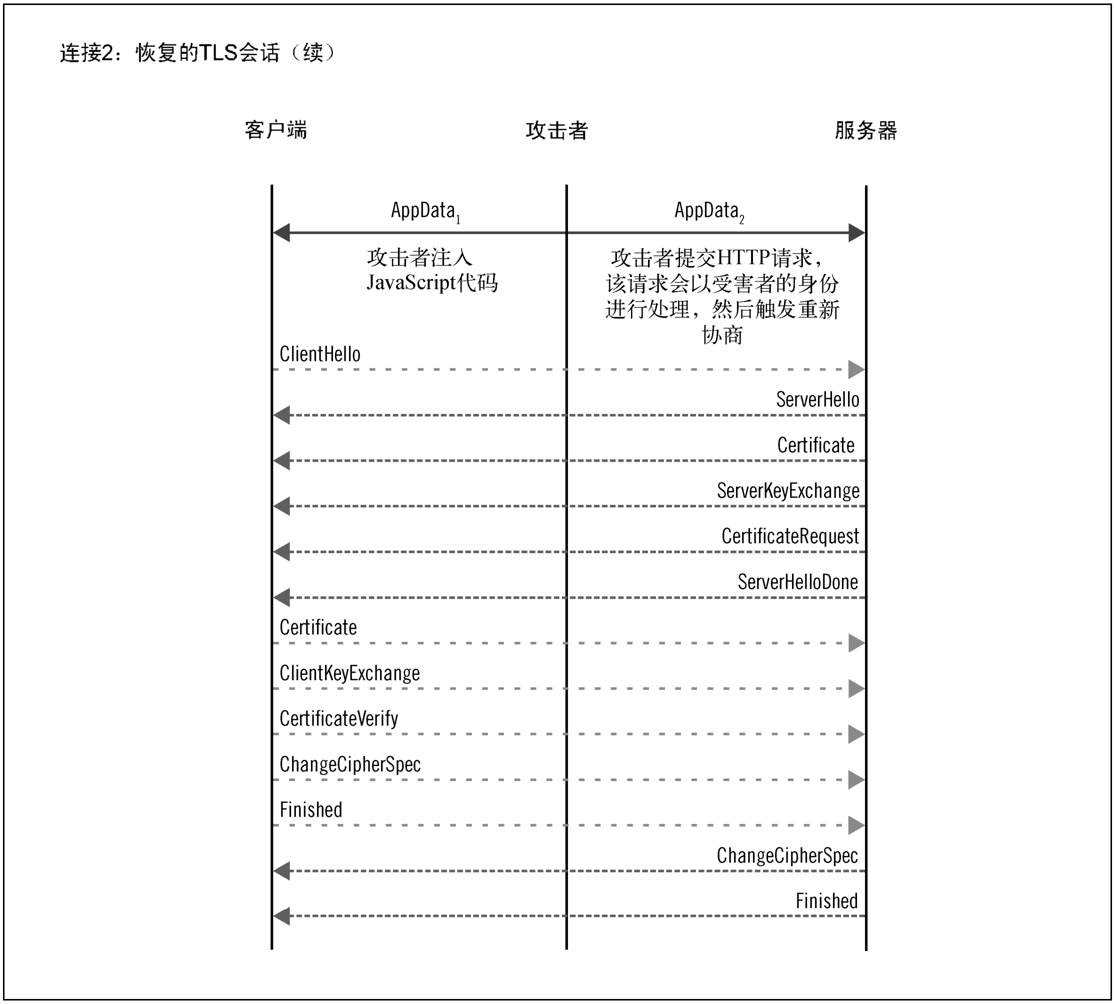


图7-10　三次握手：仿冒身份

7.6.2　影响

三次握手攻击演示了我们认为安全的TLS连接是如何被攻破的。在重新协商之前，发往目标服务器的数据是由攻击者发出的，之后的数据是由经过身份验证的受害者发出的，而目标服务器却无法进行区分。这种攻击机会类似于原始的不安全重新协商漏洞。最简单的攻击方法就是在目标网站上以受害者身份执行一个请求，例如转账。

然而，这个攻击向量不是很容易被利用。第一，攻击者要能发现应用程序中合适的进入点，并为每个请求设计负载。第二，因为在重新协商之后丢失了数据可见性，就无法了解到攻击的结

果并在同一个连接上展开后续攻击。攻击者可以展开另外的攻击，但是在TLS层上进行此操作是令人沮丧的，并且非常缓慢。

这里有另外一个可能更加危险的攻击向量。因为攻击者可以在重新协商之前向两边发送任意数据，攻击者就可以完全控制受害者的浏览器。毕竟，受害者是在访问攻击者制造的恶意网站。这允许攻击者向受害者的浏览器中注入恶意JavaScript脚本。在经过重新协商和身份验证之后，恶意脚本可以从浏览器向目标网站提交不受限制的HTTP请求（这些请求全部都在受害者的身份之下）并且可以随意获取结果。

一般来说，浏览器不允许一个网站向其他网站提交数据。在这种情况下，所有的通信都发生在攻击者的恶意网站的上下文中。数据之后才被转到目标网站，所以从浏览器的角度来看，数据提交到了同一个网站。

上述第二种攻击向量本质上是一个钓鱼攻击，并使用了三次握手攻击来处理要求客户端身份验证的问题。这是一个非常有力的攻击形式，只受限于攻击者的编程水平以及尽可能长时间地将受害者留在恶意网站上的能力。

7.6.3 先决条件

三次握手攻击十分复杂，只能在某些场景下成功实施。在利用这些漏洞之前，有两个方面的问题需要处理。

第一个是这种攻击只能用在要求使用客户端证书的网站上。如果不是这样，就无法进行身份仿冒。第二个方面更有趣。因为攻击是一种钓鱼的形式，受害者必须愿意在某个他们不常使用的网站上使用他们的证书。我情愿认为这种情况不太可能发生，但是实际上并非如此。

关于如何将用户引导到恶意网站，使用社会工程学方法或者电子邮件总是能成功，这与其他的钓鱼攻击类似。基于攻击者的位置，他也可能将明文的HTTP请求重定向到恶意网站。然而，这可能会引起用户的警觉，因为他们将毫无防备地跳转到一个陌生的网站。

基于很少网站使用客户端证书校验，实际发生的三次握手攻击的量并不大，这一点与原始的重新协商问题是不一样的。从另外一个方面来说，使用客户端身份验证的网站都是信息十分敏感的业务，所以这种攻击手段一般不会在微不足道的犯罪场景下使用。

7.6.4 缓解方法

引发三次握手攻击的核心问题在于TLS协议，这使得TLS协议是解决问题的最佳层面。对协议的相关调整工作正在进行，新的方法在握手和主密钥[①]以及会话恢复之间进行了更强的绑定[②]。

短期来看，浏览器厂商采取了一些措施，他们将软件修改成如果重新协商之后的证书与之前

① TLS Session Hash and Extended Master Secret Extension，https://datatracker.ietf.org/doc/draft-bhargavan-tls-session-hash/（Bhargavan等，2014年4月）。

② TLS Resumption Indication Extension，http://datatracker.ietf.org/doc/draft-bhargavan-tls-resumption- indication/（Bhargavan等，2014年4月）。

的不同，则断开连接。同样，退化的DH公钥将不再被接受。当然，这些手段都是在比较新的浏览器版本中才有。老的IE浏览器也是安全的，因为Microsoft对系统的SSL库进行了修正，不仅仅是针对浏览器。

除了浏览器的改进之外，还有一些残留的攻击向量可以在某些情况下被利用来进行攻击（当不使用证书的时候）：SASL、PEAP和Channel ID。除非协议进行修改，否则这些都无法解决。

如果可能，我建议你在服务器端采取一些措施来最小化风险。最新的浏览器没有问题，但是依然有很多用户在使用老版本的浏览器，这就容易被攻击。考虑以下这些方法。

- 对所有访问要求客户端证书

  如果全站的所有访问均要求客户端证书，那攻击者首次对目标网站的访问也需要提供证书。具体的情况还取决于攻击者是否能很容易获取到客户端证书，但是仅考虑此方法本身，是可以降低攻击风险的。

- 禁用重新协商

  攻击展开的强依赖是要进行重新协商。然而，重新协商同样是与客户端身份验证一起使用的。例如，某个网站允许任何人访问首页，但是要求只有经过身份验证才能访问某个子目录。如果这种业务结构可以改变，那么就不会有重新协商，也不会产生攻击了。

- 只启用ECDHE套件

  ECDHE套件不受此攻击威胁。基于所有现代浏览器都支持ECDHE，如果用户群体较小并且不使用老旧浏览器（主要是Android 2.x和Windows XP上的IE），禁用易受攻击的密钥交换（DHE和RSA）也许是另外一个不错的选择。不过这个方法不适合用户群多样性强的情况。

7.7　POODLE

2014年10月，Google安全团队公布了POODLE（Padding Oracle On Downgraded Legacy Encryption），这是一个SSL 3中的漏洞，可以让攻击者破解小段的加密数据。①

造成这个问题的根本原因是CBC模式在设计上的缺陷，具体来说就是CBC只对明文进行了身份验证，但是却没有对填充字节进行完整性校验。这使得攻击者可以对填充字节进行修改并利用填充预示来恢复加密内容。我在本章前面部分讨论过这个话题。如果你对填充预示攻击还不了解，我建议你在继续阅读本节之前，先阅读7.4节。

让POODLE攻击成为可能的原因是SSL 3中过于松散的填充结构和校验规则，但是这些问题在TLS 1.0和以后的版本中被修复了。你可以在图7-11中发现两者的不同之处。

① This POODLE bites: exploiting the SSL 3.0 fallback，http://googleonlinesecurity.blogspot.co.uk/2014/10/this-poodle-bites-exploiting-ssl-30.html（Google Security Team，2014年10月14日）。

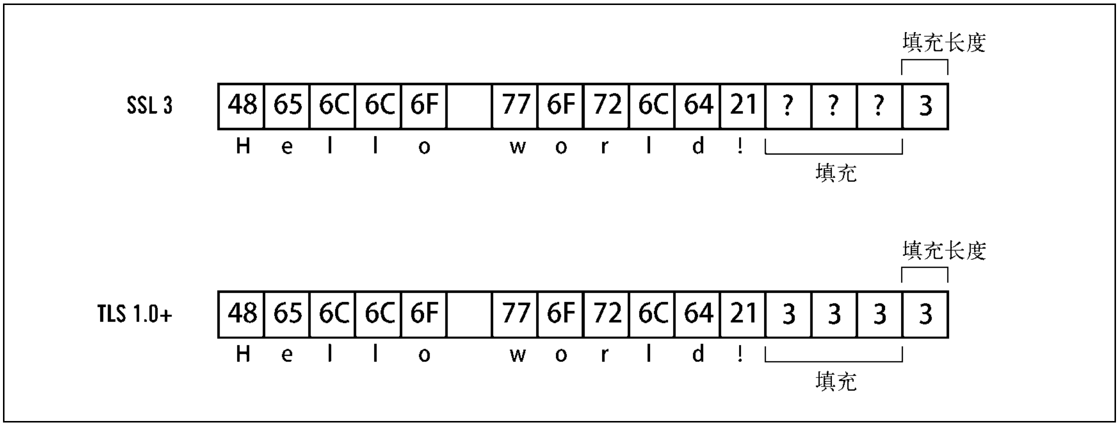


图7-11 SSL 3.0和TLS 1.0+中填充的对比

SSL 3的填充方式是保留密文的最后一字节，并填写填充长度，但是并没有规定具体的填充数据应该如何构建。重要的是，没有规定校验规则来确认填充数据是否被篡改过。也就是说，SSL 3的填充具有不确定性。在TLS 1.0以及之后的版本中，发送方必须在填充空间中填满与填充的长度值相同的字节。接收方在解密之后立即校验填充。如果填充长度与所有的填充字节不同，会认为整个加密块无效而将其丢弃。

进行POODLE攻击，攻击者必须能在不影响MAC或者改变其他明文的情况下修改填充。这就是说要重新安排明文数据使得最后一个完整的加密块都是填充字节。这是必要的，因为攻击者看见的都是密文，无法直接修改填充字节。因为分组加密的特点，即使在密文中任何一位的修改，都会使得几乎一半的数据在解密之后发生变化。然而，如果最后整个数据块都是填充数据，攻击者就可以进行任意的修改而不会导致MAC校验失败。

我们来假设攻击者进行了一个小的修改。因为这个修改会导致解密之后产生大量变化，解密之后的内容会变成随机内容。然而，由于SSL 3中松散的填充校验，填充中的字节并不会被检查，只有最后的填充长度字节必须正确。在实际中，这意味着每256次探测，就会有一次被服务器接受，无论其中的修改是什么。

攻击者可以利用这个特性来干什么？他无法直接破解出任何明文，但是他知道他的修改在什么时候被接受了，最后一字节在加密之后得到了正确的填充长度。这字节的值和填充的最大程度相等，即对于16字节块大小来说是15（例如AES），对于8字节块大小来说是7（例如3DES）。因此，他得到了一个预示：他刚刚找到了实施攻击的办法。为此，他可以检测CBC结构的细节（如图7-12所示）。

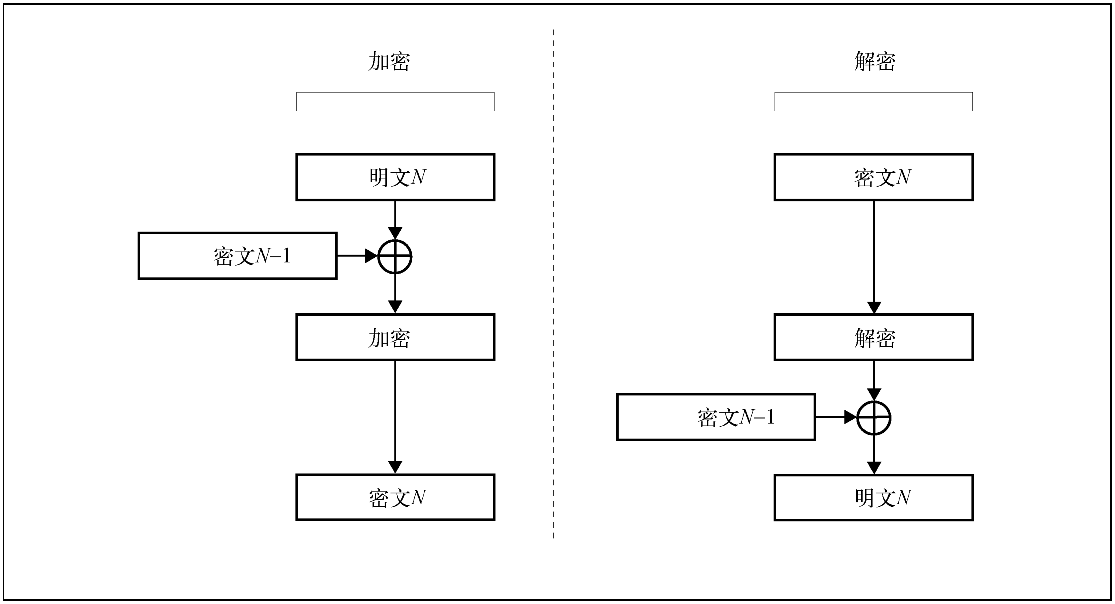


图7-12　CBC加密和解密

在进行加密之前，一块数据首先要与IV进行异或，IV本质上就是一块随机数据。然而，虽然IV是随机的，但是攻击者可以知道从第二个块开始的IV值。[①]

为了简单明了，我们把焦点集中到攻击者的目标明文上：

$$E(P_i[15] \oplus C_{i-1}[15])=C_i[15]$$

虽然两个加密字节都是已知的，但是攻击者也无法还原出明文，因为他不知道密钥。

现在来考虑逆向过程。首先进行解密操作，之后对解密的结果使用正确的IV（前一个加密块的密文）进行异或操作。同样，攻击者知道密文，因为这可以从网络上观察到。因为SSL 3的缺陷，他同时知道数据块的最后一字节异或之后的值是15。这就足够进行攻击了：

$$D(C_i[15]) \oplus C_{n-1}[15] = 15$$
$$D(C_i[15]) = 15 \oplus C_{n-1}[15]$$
$$P_i[15] \oplus C_{i-1}[15] = 15 \oplus C_{n-1}[15]$$
$$P_i[15] = 15 \oplus C_{n-1}[15] \oplus C_{i-1}[15]$$

因此，当猜测正确的时候，攻击者仅需要两次异或操作就可以获取到1字节的明文数据。

① 在SSL3中，连接上的第一个IV块是从主密钥导出的，攻击者不知道它的值。然而，所有后续的IV块都是可知的，因为它们在CBC的链中，当前块的IV就是前一个加密块的内容。

7.7.1 实际攻击

在这种情况下，与最近发生的针对SSL/TLS的类似攻击一样，POODLE也要求进行复杂的设置：攻击者必须对受害者的浏览器有足够的控制权才能向目标服务器提交任意数据。此外，他还得能控制网络以便与浏览器中的操作相互协调。这种技术最早在BEAST攻击中被发明，后来被其他攻击方法所使用。

每一次攻击尝试都包括由浏览器发出的原始请求和攻击者替换掉的密文的最后一字节。这种行为会不断重复直到猜测正确，此时已经可以破解1字节的明文。攻击者于是开始进行下一字节的攻击。

在每个请求内，攻击者必须要影响待破解字节的位置并且控制填充以使得填充能消耗掉最后一整个加密块。这也就是说需要有在待破解数据前后都注入数据的能力。在实际中，这个可以通过使用POST请求并操作URL和请求体来实现。

下面这个例子展示了一个在加密之前的TLS记录：应用层数据在最开始处，之后跟随的是MAC（M），然后是填充（padding，P），最后是填充长度。带破解数据是会话ID，存放于名为JSESSIONID的Cookie里。就像要求的那样，Cookie的第一字节是加密块数据的最后一字节。

```
00000000  50 4f 53 54 20 2f 61 61 61 61 20 48 54 54 50 2f  |POST /aaaa HTTP/|
00000010  31 2e 30 0d 0a 48 6f 73 74 3a 20 65 78 61 6d 70  |1.0..Host: examp|
00000020  6c 65 2e 63 6f 6d 0d 0a 43 6f 6e 74 65 6e 74 2d  |le.com..Content-|
00000030  4c 65 6e 67 74 68 3a 20 31 32 0d 0a 43 6f 6f 6b  |Length: 12..Cook|
00000040  69 65 3a 20 4a 53 45 53 53 49 4f 4e 49 44 3d 42  |ie: JSESSIONID=B|
00000050  33 44 46 34 42 30 37 41 45 33 33 43 41 0d 0a 0d  |3DF4B07AE33CA...|
00000060  0a 30 31 32 33 34 35 36 37 38 39 41 4d 4d 4d 4d  |.0123456789AMMMM|
00000070  4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d  |MMMMMMMMMMMMMMMM|
00000080  50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 0f  |PPPPPPPPPPPPPPP.|
```

得到正确的URL和请求体长度对于第一次尝试来说是不太容易的，因为填充隐藏了明文的真实长度。这可以通过每次改变1字节的负载长度并观察加密后的变化来解决。例如，假设提交的数据如前例所示，从URL中移除1字节将会将填充长度从15更改为0，从而将总体已加密文本的长度降低了一整块的长度（16字节）。一旦这个现象发生了，攻击者就可以知道确切的填充长度。

在这之后，事情就变得容易了。猜测成功之后，通过将URL减少1字节，并将请求体增加1字节，从而使得Cookie中下一字节满足攻击条件。

因此，整个攻击将按照以下方式展开：(1) 攻击者以一个大的URL和一个小的请求体展开攻击；(2) 缩短URL，每次1字节，直到找到正确的填充长度；(3) 提交足够多次的猜测以破解1字节的数据；(4) 通过同步改变URL和请求体的大小来循环破解其余字节。

一个典型的会话ID可能包括16~32字节，但是我们假设攻击者需要破解100~200字节，因为他不知道Cookie的精确长度。每256个请求就可以破解1字节的速度，这将是本章中所讨论的密码学攻击中效率最高的攻击之一了。

7.7.2 影响

鉴于SSL 3已经有20个年头了，并且几乎所有服务器都支持TLS 1.0，你可能会认为POODLE不会造成太大的危害。毕竟从SSL 3开始协议本身存在内在的防御降级攻击的办法。那么是否每个人都使用最好协议来进行握手呢？不幸的是，这只是理论上成立。在实际中，所有的主流浏览器都设计成了在连接或者TLS握手失败的时候进行协议降级。这样一来，当POODLE被公布的时候，几乎所有的浏览器都可以被降级成SSL 3，即使网站支持更高版本的协议。我在6.6节中详细介绍了这个问题。

伴随着协议降级，POODLE攻击变得相对容易执行，但它同时也要求攻击者具备足够的经验并且距离受害者较近。与其他同时使用控制浏览器和网络手段的攻击方法一样，POODLE攻击的目标就是窃取小但是有价值的信息，典型的就是Cookie值或者密码。这个门槛使得POODLE攻击一般只会针对高价值的网站展开。一次成功的会话Cookie窃取可以让攻击者仿冒受害者的身份并使用此身份展开一切其他的行动。

POODLE TLS

虽然TLS 1.0在填充结构方面进行了改进，但是在2014年12月被发现POODLE也会影响使用TLS的应用程序和设备①。问题没有出在协议上，而是出在了实现上。看起来是一些开发人员在进行SSL 3到TLS的转换的时候，没有遵守协议规定的填充要求，使得他们的实现容易受到POODLE攻击的威胁。至少在几个案例中，在很多产品使用的硬件加速卡中发现了漏洞。

根据SSL Pulse的数据，在2014年12月，有大约10%的服务器受到POODLE TLS威胁。在威胁被发现几个月之后，我们仍然不知道是否已经了解了全部受威胁的产品②。最好的做法就是使用某个SSL/TLS服务器测试工具对你的基础设施进行检测。

7.7.3 缓解方法

面对一个老旧协议的严重漏洞，浏览器厂商决定不采取任何应对措施，反而希望能利用此机会加速SSL 3的退役。最初的方法是减小攻击面。Chrome、Firefox和IE禁止回退到SSL 3。Safari保留了这个回退，但是在这种情况下禁用了所有的CBC模式的算法。从那之后，Chrome和Firefox都禁用了SSL 3（版本分别是40和34）。2015年4月，IE11也通过升级禁用了SSL 3。

> **注意**
> 一个新的防止协议降级的方法正在标准化中。这个方法使用了一个特殊信号套件值

① Poodle Bites TLS，http://blog.ivanristic.com/2014/12/poodle-bites-tls.html（Ivan Ristić，2014年12月8日）。

② There are more POODLEs in the forest，https://vivaldi.net/en-US/blogs/entry/there-are-more-poodles-in-the-forest（Yngve Pettersen，2015年7月14日）。

> TLS_FALLBACK_SCSV，浏览器可以通过它通知服务器自己被降级了。可以理解这个套件的服务器就可以基于此对连接进行阻断，从而防止攻击发生。Google从Chrome 33开始支持这个特性。[①]Firefox从35版本开始支持[②]。这是一个针对协议降级攻击的长期手段，但是与解决POODLE关系不大。支持这个特性的浏览器都已经禁用了SSL 3，而只支持SSL 3的老旧浏览器不支持此特性。此外，IE浏览器团队指出他们不会支持这个特性[③]。

最好的防御这个攻击的方法就是彻底禁用SSL 3。很多公司，包括一些很大的公司（比如Amazon）已经在这么做了。在某些情况下，这可能会影响用户访问你网站的可用性。例如，在Windows XP上的IE6默认不支持TLS 1.0，虽然可以通过配置启用。如果你认为你无法禁用SSL 3，那么可以通过只使用RC4来避免POODLE攻击，但是这种行为又带来了另外的糟糕问题，因为RC4已经超期服役很久了。

应该注意的是，虽然所有支持SSL 3的服务器会受到POODLE威胁，但是由于成功的攻击需要进行交互，这就意味着在客户端方面，只有浏览器才可能受到威胁。其他的客户端和非HTTP的协议可能不会有任何问题。然而，你不应该让这种情况减慢你剔除SSL 3支持的步伐。业界对于废弃SSL 3有很高的呼声。例如，PCI委员会计划在PCI DSS 3.1中完全废弃SSL 3[④]。一个新的禁止SSL 3使用的RFC也已经准备公布[⑤]。

7

7.8 Bullrun

Bullrun（或者写作BULLRUN）是美国国家安全局（National Security Agency，NSA）运行的一个机密计划的代号。该计划的主要目的是使用任何可能的手段破解加密通信。一个最可能成功的办法就是直接针对服务器进行攻击。如果你能获得服务器的私钥，也就不用再研究加密通信如何进行破解了。对我们来说很感兴趣的一点是，此计划的一种手段是降低产品和安全的标准。下面是一个泄露的绝密文件中的一份预算草案中的一段话[⑥]。

> 对商用公钥密码技术的策略、标准和规范施加影响。

① TLS Symmetric Crypto，https://www.imperialviolet.org/2014/02/27/tlssymmetriccrypto.html（Adam Langley，2014年2月27日）。

② The POODLE Attack and the End of SSL 3.0，https://blog.mozilla.org/security/2014/10/14/the-poodle-attack-and-the-end-of-ssl-3-0/（Mozilla Security Blog，2014年10月14日）。

③ Internet Explorer should send TLS_FALLBACK_SCSV，https://connect.microsoft.com/IE/feedback/details/1002874/internet-explorer-should-send-tls-fallback-scsv（IE Feedback Home，检索于2015年3月16日）。

④ PCI SSC bulletin on impending revisions to PCI DSS, PA-DSS，http://training.pcisecuritystandards.org/pci-ssc-bulletin-on-impending-revisions-to-pci-dss-pa-dss-assessor（PCI Council，2015年2月13日）。

⑤ RFC 7568: Deprecating Secure Sockets Layer Version 3.0，https://tools.ietf.org/html/rfc7568（Barnes等，2015年6月）。

⑥ Secret Documents Reveal N.S.A. Campaign Against Encryption，http://www.nytimes.com/interactive/2013/09/05/us/documents-reveal-nsa-campaign-against-encryption.html（《纽约时报》，2013年9月5日）。

《纽约时报》称，NSA每年会花费大约2.5亿美元来做这些事情。英国的GCHQ[①]也有自己类似的计划来对加密进行破解，计划代号Edgehill[②]。

作为一个最主要的安全协议，TLS很明显是这些计划的目标。Bullrun的公开披露使得很多人以完全不同的眼光来看待协议标准的开发过程。如果连制定标准的人我们都不信任的话，我们如何能信任这些标准？

双椭圆曲线确定性随机位生成器

双椭圆曲线确定性随机位生成器（dual elliptic curve deterministic random bit generator，Dual EC DRBG）是一个伪随机数生成器（pseudorandom number generator，PRNG）的算法，被国际标准化组织（International Organization for Standardization，ISO）在2005年的时候标准化为ISO 18031，被美国国家标准与技术研究院（National Institute of Standards and Technology，NIST）在2006年标准化[③]。

在2007年，两个研究人员发现该算法中存在一个可能的后门[④]，但是他们的发现没有引起足够的关注。

当2013年9月，Bullrun计划公诸于世的时候，Dual EC DRBG被暗示为是NSA留的后门。同月，NIST发布公告来废除该算法[⑤]。

> NIST强烈建议，停止对SP 800-90A的安全考虑和重新签发的决议，同时，在2012年1月份版本的SP 800-90A中定义的Dual EC DRBG算法，也不再继续使用。

在2013年，路透社报道NSA向RSA Security公司支付了1000万美元款项，使得RSA公司将Dual EC DRBG作为他们的TLS实现（BSAFE中的默认随机数生成算法）[⑥]。其他的TLS实现也将Dual EC DRBG作为可选项（主要是为了FIPS 140-2验证），但是据我们所知，并没有默认启用。OpenSSL中的相关实现被发现存在问题，因而实际上无法使用[⑦]。

也许你会问，这将会如何影响TLS？在密码学领域，所有安全性都依赖于随机数生成的质量好坏。历史上我们遇见过很多在这里出问题的实现，具体可以参考6.2节。如果你能攻破某人的随机数生成器，那么就有机会攻破其他任何东西。TLS协议要求客户端和服务器各自发送28字节

---

① 英国政府通讯总部，类似NSA的情报机构。——译者注

② Revealed: how US and UK spy agencies defeat internet privacy and security，http://www.theguardian.com/world/2013/sep/05/nsa-gchq-encryption-codes-security（《卫报》，2013年9月6日）。

③ Dual_EC_DRBG，https://en.wikipedia.org/wiki/Dual_EC_DRBG（维基百科，检索于2014年4月3日）。

④ On the Possibility of a Back Door in the NIST SP800-90 Dual Ec Prng，http://rump2007.cr.yp.to/15-shumow.pdf（Shumow和Ferguson，2007年8月）。

⑤ SUPPLEMENTAL ITL BULLETIN FOR SEPTEMBER 2013，http://csrc.nist.gov/publications/nistbul/itlbul2013_09_supplemental.pdf（美国国家标准与技术研究院，2013年9月）。

⑥ Exclusive: Secret contract tied NSA and security industry pioneer，http://www.reuters.com/article/2013/12/20/us-usa-security-rsa-idUSBRE9BJ1C220131220（路透社，2013年12月20日）。

⑦ Flaw in Dual EC DRBG (no, not that one)（Steve Marquess，2013年12月19日）。

的随机数据作为握手的一部分，这些数据会参与主密钥的生成，而主密钥用来保护整个TLS会话。如果你能在随机数生成器里安插后门，那这28字节可能足够用来揭示生成器的运行状态，并在随后协助破解TLS会话。

2014年，研究人员演示了Dual EC DRBG确实可以当成后门使用[①]，虽然他们无法证明这是有意的后门。与此同时，一个非标准的TLS扩展，在NSA的要求下，在BSAFE中得到了实现，用于在一个TLS连接上从随机数生成器中暴露更多的数据[②]。

随着更多的随机数据暴露给攻击者，攻破TLS连接会容易65 000倍。

① On the Practical Exploitability of Dual EC in TLS Implementations，http://dualec.org/（Checkoway等，2014年）。

② Extended Random，https://projectbullrun.org/dual-ec/ext-rand.html（projectbullrun.org，检索于2014年7月16日）。

第8章 部　　署

前面各章讲的都是理论和背景知识，本章将把所有内容综合在一起，给你一些比较高层的建议，即关于TLS服务安全部署你应该了解的所有知识。可以这么理解，本章是整本书的一个地图，当你阅读到一些地方并需要了解细节时，可以参阅书中相关章节的具体内容。当你觉得已经掌握了所有知识后想加以实践时，可以阅读后面关于平台配置实践的章节。

学习本章时最好与第9章一起阅读。本章会有很多性能方面的建议，而第9章则是更具体的配置和说明，可以快速应用在你自己的网站部署中。

8.1 密钥

私钥是TLS安全的基石，为私钥选择恰当的密钥算法和长度，TLS可以提供高强度的安全性（在现有计算条件下需要很多年才能被破解）。实际上，抛开数学方面的考虑（位数越高越安全），TLS最大的弱点在于密钥的管理，或者说如何保护私钥的私密性。

8.1.1 密钥算法

目前TLS支持3种算法，但实际上只有RSA这一种被广泛使用；DSA已经被废弃，而ECDSA在未来几年内有望被广泛使用。

- DSA
  DSA算法很容易被排除：因为DSA的密钥长度最大只能到1024位（IE浏览器也不支持更高强度），这个位数根本无法确保安全性，所以没有人会在TLS实际应用中使用DSA算法，与所有人背道而驰的结果是让你陷入兼容性问题中。
- RSA
  RSA算法是最常见的一种选择，基本上所有的TLS部署都会支持RSA算法。但是，RSA在2048位（最小位数）的密钥下，比ECDSA密钥在安全性上更弱并且性能更差。更糟的是，增加RSA密钥长度是性能消耗的增加并不是线性的，如果你觉得2048位的加密强度不够，需要使用更高的位数（比如3072位）的RSA密钥时，在性能上就会有极大幅度的下降。
- ECDSA
  ECDSA算法是未来的选择。256位的ECDSA密钥有128位用于安全加密上，相对而言，2048

位的RSA密钥只有112位是真正用于安全加密的。不仅如此，在这个加密强度下，ECDSA比RSA算法快2倍；如果是与3072位的RSA密钥在相同加密强度下相比，ECDSA性能要快6倍以上。[①]

因为椭圆曲线（elliptic curve，EC）算法是最近才被加入到TLS的安全体系中，所以ECDSA的问题在于：不是所有用户端都支持这种算法。新的浏览器都支持ECDSA，但是一些老版本的浏览器是不支持这个算法的。你可以通过同时部署RSA和ECDSA的密钥来解决对新老浏览器的兼容，但并不是所有服务程序都能提供这种配置方式，另外这种方案也需要额外的工作来同时维护两套密钥和证书。因此，就现状而言，ECDSA的最佳使用场景是用于部署追求最高性能的TLS服务系统。未来，随着我们对安全的日益重视，ECDSA也会变得越来越重要。

8.1.2　密钥长度

在密钥长度方面，大部分系统部署只需要2048位RSA密钥或者256位ECDSA密钥，分别提供112位和128位的加密强度。也就是说，大部分系统部署可以使用算法的最低安全密钥长度，因为即使如此系统也足以满足用户的安全需求。

从长期的安全保护来说，你应该使用至少128位加密强度的安全密钥。在这个条件下，256位的ECDSA密钥是最佳也最实用的选择。如果用RSA就必须使用3072位密钥，但性能上就差得太多。双密钥部署是一种不错的选择，既可以避免性能的损失，又能提供最好的浏览器支持：现代浏览器（希望你的客户群体都属于这一类）可以使用ECDSA密钥，而其他用户可以使用较弱的RSA密钥，或者是接受强RSA密钥的性能损失。

> **警告**
> 如果你还在使用低于112位加密强度的密钥，比如1024位的RSA密钥，请立即更新你的密钥。低于112位强度的密钥是不安全的，尤其是512或者768位的RSA密钥，只需要很少的计算资源就可以破解。据评估，破解1024位的RSA密钥的大约成本是100万美元。

在选择密钥长度时，可以参考以下几条来考量：(1) 在当前的时间点上是否已足够安全？(2) 在密钥过期之前（有效期内）数据是否一直能保持安全性？(3) 在密钥过期后，数据还能保持多久的安全性？

8.1.3　密钥管理

我们已经讨论了很多密钥长度方面的安全性，但实际上密钥管理问题更可能对你的系统安全

① 性能对比仅供参考，因为算法的实现在不断改进，例如OpenSSL的新版本对常用场景中的椭圆曲线加密进行了性能优化。

产生严重的影响。有充分的证据表明，大部分成功的攻击都是绕过了加密环节，而不是直接破解出加密内容。如果有人能侵入你的系统并拿到密钥或者以某种方式诱使你公开了密钥，那他们也就不再需要烦恼于如何暴力破解你的加密数据了。

- 保证私钥的私密性

  把你的私钥作为最重要的资产来保护，在条件允许的范围内，只给最少的员工提供访问权限。有些CA会提供生成私钥的功能，务必不要使用这些功能；事实上这些CA有必要提高一下自己的安全意识。私钥的名称已经说明一切，无论什么情况下，都必须保证私钥的私密性。

- 仔细选择随机数生成器

  密钥的安全性依赖于计算机上随机数生成器（random number generator，RNG）的质量，密钥的生成也离不开随机数生成器。有一种常见的场景是在服务器刚安装完并重新启动时完成了密钥的生成，但是在这个点上服务器可能还没有足够的随机信息（熵）用于生成强密钥。更好的密钥生成办法是：使用一个独立的（离线的）服务器，并且确保服务器上已经部署了一个强壮的RNG。

- 保护密钥的密码

  密钥在生成时都需要一个密码用于密钥创建，同时这个密码也用于保护密钥的内容。假如你的密钥备份被窃取，也无法被直接用于攻击。这同样也有助于在计算机之间复制密钥时（直接复制或者使用U盘）避免密钥内容的泄露。要知道，从现代的文件系统里真正删除数据变得越来越困难了。

- 不要随意共享密钥

  共享密钥是非常危险的。任意一个系统被攻破，被窃取的密钥就可以直接用于攻击其他系统，就算这些系统的证书不同也无法幸免。让不同系统使用不同的密钥，并通过限制密钥的访问，只提供给真正需要的用户使用，才能保障内部访问控制的安全。

- 定期更新密钥

  你有义务维护好私钥的使用。在创建密钥时注意不要使用过长的有效期；发生安全事件或者核心员工离职时，应该更改密钥并且生成新的证书；当你生成一个新的密钥时，务必把旧的密钥完全清除干净。尤其是对于不支持前向保密的系统来说，你的密钥可以用于解密以前所有的通信。通过安全地清除旧密钥，可以保证你的对手今后再也无法用它来对付你，即使他们还保存着你的历史通信记录。正常情况下，你应该每年更新一次密钥；而对于不支持前向保密的系统（强烈建议改进），需要更频繁地更新密钥，例如每季度一次。

- 安全存储密钥

  找一个安全的地方保存密钥副本。对于普通服务器而言，丢失密钥并不是什么大事情，因为你可以重新生成一个新密钥；但如果密钥是中间或者私有CA用来签发证书的或者用于钉扎，就完全不一样了。

  这种场景下使用防篡改硬件（如果成本不是问题）来创建和保存私钥是最安全的方案。

这种设备被称为硬件安全模块（hardware security module，HSM）。使用HSM时，私钥永远不能脱离HSM，甚至无法用物理方式提取。目前市场上已经有HSM服务[①]提供，如果你对安全非常关心并考虑使用HSM，注意使用云平台的HSM服务目前还是不靠谱的。鉴于我们所知道的高科技监听手段[②]，即使把HSM服务部署在内部区域，找到一个不会在系统中部署后门的可靠制造商也很困难。毕竟你一定不想花很多钱买来一个HSM设备，最后却发现密钥可以从这个设备上被他人复制走。

8.2　证书

在这一节中，我们会来讨论证书选择的内容。证书选择需要做很多方面的考虑：包括使用什么类型的证书，每个证书需要包含哪些域名，使用哪家CA来签发证书，等等。

8.2.1　证书类型

证书主要有三种类型：域名验证（domain validated，DV）、组织验证（organization validated，OV）和扩展验证（extended validation，EV）。DV证书的签发是自动的，通常也是最便宜的，一般的网站使用DV证书就足够了。OV证书需要验证域名拥有者的公司信息，并且在证书中包含公司信息。尽管如此，浏览器实际上并不区分OV和DV证书，也不会展示出证书包含的所有信息。

EV证书在以下几个方面与DV和OV证书不同：(1) EV的验证过程符合CAB论坛标准；(2) 公司信息显示在浏览器地址栏并突出显示为绿色；(3) 浏览器对EV证书的证书吊销支持更好一些。从安全角度讲，EV证书并无多少改善，但它们确实能给用户更好的体验，对于某些业务来讲，这种体验可能很有价值。

> **注意**
>
> 如果你计划采购EV证书，Chrome浏览器要求2015年1月1日之后签发的EV证书都必须支持**证书透明度**（certificate transparency，CT）特性，所以在购买前请务必咨询你的CA厂商是否支持CT，以确保证书可以被Chrome正确识别。

8.2.2　证书主机名

证书的主要作用就是为保障用户安全顺畅地访问域名建议合适的信任机制。在互联网上，用户经常会收到一些证书域名不匹配的警告，这往往是由于使用的证书不能匹配该域名的不同变体（比如证书域名是www.example.com，无法匹配example.com域名）。

为了避免上述问题，请遵守一个简单的规则：只要有一个DNS解析指向你的TLS服务器，就

① AWS CloudHSM，https://aws.amazon.com/cloudhsm/（亚马逊Web服务，检索于2014年5月16日）。

② Photos of an NSA “upgrade” factory show Cisco router getting implant，http://arstechnica.com/tech-policy/2014/05/photos-of-an-nsa-upgrade-factory-show-cisco-router-getting-implant/（Ars Technica，2014年5月14日）。

务必保证你的证书包括了这个DNS域名。我们无法控制用户在浏览器地址栏中输入网址的方式，也无法控制他们连接到我们网站的途径；我们唯一能保障的是证书能够覆盖所有有效的域名。从我的经验来看，有些CA签发证书时会自动包含域名的不同变体，但也有些CA不具备这个功能。

> **注意**
> 另一个常见的问题是有些网站会同时把明文服务（http）和安全服务（https）部署在同一个IP地址上。当有人使用https://前缀来访问明文服务的网站域名时，就会收到证书名称不匹配的警告，用户忽略警告后服务又会被错误地转发到安全服务上。解决这个问题的最好办法是把明文服务独立部署到其他IP地址，并关闭这个IP的443端口。

8.2.3　证书共享

证书共享大致分为两种类型。第一种：你可以签发一个包含所有需要域名的证书（例如，www.example.com、example.com和log.example.com）。第二种：你可以签发一个泛域名证书，可用于所有关联的子域名（例如，签发一个*.example.com和example.com的泛域名证书）。

多域名证书的优点是降低了维护成本并且可以让多个https网站共用一个IP地址。这种证书在CDN上使用很普遍，因为CDN经常会需要对很多不同的外部用户提供服务。

原则上，使用多域名证书并没有什么问题，但前提是它不会降低你的系统安全性，而现实是这种证书的使用确实会对安全性有影响。从本质上来讲，当多个域名共用一个证书时，事实上它们也就共用了一个相同的私钥。因此，当你的一组网站是由不同团队运营或者根本各不相关时，就不应该对这些网站使用多域名证书。因为如果其中一个网站被攻击，被窃取的私钥就可以用来攻击所有共用这个证书的其他网站，另外在发生攻击后，你也需要对这组网站全部更新一遍密钥。

更重要的是，共享一个证书会在应用层面上将一组业务捆绑在一起：任意一个站点的漏洞，都可以用来攻击这组业务的所有其他网站。由于有这些安全隐患，最好在实践中避免使用多域名证书。此外，在不同服务之间共享TLS会话也有类似的问题，你可以在6.8.1节中找到关于这个问题的更全面的讨论。

8.2.4　签名算法

为了保证证书的有效性，CA在签发证书时会对证书进行数字签名。数字签名的安全性依赖于两个方面：一个是CA的私钥长度，另一个是散列算法的强度。一般来讲，CA私钥的强度都是可以保证的，但目前签名最常用的散列算法（SHA1）的安全性是有问题的。虽然SHA1在设计上提供了80位的强度，但实际使用中往往只能达到61位。

自从2009年MD5签名算法被完全破解后，业界就已开始着手SHA1算法的迁移。2013年，Microsoft率先提出将于2017年1月开始停止接受SHA1证书①，各大CA厂商也随之开始纷纷改用

① SHA1弃用策略，http://blogs.technet.com/b/pki/archive/2013/11/12/sha1-deprecation-policy.aspx?Redirected=true（Windows PKI Blog，2013年12月12日）

SHA256作为其默认的散列签名算法。2014年9月，Google宣布将在2014年底对SHA1证书提示安全警告。从Chrome 42版本后，Chrome浏览器对2016年内到期的SHA1证书会提示警告，对2017年后到期的SHA1证书则直接提示错误。这些措施都大大促进了业界SHA1证书的迁移。①

对于新申请的证书，要确保使用SHA256或更好的签名算法。申请证书前你需要提前与CA确认好签名算法，因为签名算法的选择不是在你提交的CSR里设定的，并且需要确认CA签发证书的整个证书链都不能使用SHA1签名（根证书签名例外）。对于现有的证书，如果有效期在2016年之前，你可以继续使用，否则应该立刻采取措施更新证书。

> **注意**
> 每一次密码算法的更新，最大难题就是老客户端的兼容性问题，老客户端往往不支持新的特性。对于SHA256来说，最大的问题是没有升级到SP3的Windows XP用户②，以及Android 2.3版本之前的移动设备③，二者都不支持SHA256算法。
> 因此在更新证书签名算法之前，请仔细检查你的Web服务器日志，看一下这些老客户端在你的流量中还占据多大的比例。如果比例还很大，可以推迟更新证书，直到最后的时刻（2017年），以尽量避免用户流量的损失。另外还有一种方法。有一些Web服务器支持对一个站点部署多个证书的方案，如果你的Web服务器支持这个特性，那就可以做到两全其美：对老客户端部署SHA1证书，同时对其他用户部署SHA256证书。我会在13.5节中详细介绍Apache Web服务器多证书的部署方法。

8.2.5 证书链

虽然我们一直在讨论服务器证书，但实际上我们在部署TLS服务器时真正需要配置的是证书链（certificate chain）。一个证书链就是能溯源到一个可信根证书的有序证书列表。互联网上一个常见的问题就是服务器没有配置完整的证书链，根据SSL Pulse 2014年7月的数据，大约有5.9%的服务器没有正确配置证书链。④

一些UA知道如何重建不完整的证书链，有两种常见的方法：(1) 预置并缓存所有的中间CA证书；(2) UA从证书内置的父证书信息中取得链接地址并自动获取。这两种方法都不可靠，第二种方法还会降低用户访问速度，因为UA还需要从CA网站上下载证书链来验证证书的有效性。

另一个常见问题是证书链的顺序不正确，严格来讲这就是一个无效的证书。实际上，几乎所有UA都知道如何重新排序来建立正确的证书链。当然最好的办法是确保你的证书链完整并且顺

---

① Gradually sunsetting SHA-1，http://googleonlinesecurity.blogspot.co.uk/2014/09/gradually-sunsetting-sha-1.html（Google Online Security Blog，2014年9月5日）。

② SHA-256 certificates are coming，https://www.imperialviolet.org/2014/05/14/sha256.html（Adam Langley，2014年5月14日）。

③ SHA-256 Compatibility，https://support.globalsign.com/customer/portal/articles/1499561-sha-256-compatibility（GlobalSign，检索于2014年9月26日）。

④ SSL Pulse，https://www.trustworthyinternet.org/ssl-pulse/（SSL Labs，检索于2015年7月26日）。

序正确。

虽然中间证书的有效期一般很长，但总有一天还是会过期。因此当你安装一个新证书时，即使CA不变，也建议更新证书链中的全部证书，这样可以确保不会错误地部署了过期的中间证书。

从性能优化角度来看，你的证书链应该只包含所有必要的证书，既不多也不少，额外的证书（比如根证书，是不需要包含的）会降低TLS握手速度。麻烦的是有时我们还会遇到多证书链的场景。由于历史原因，存在多个可信路径是有可能的，比如一个新的CA将自己的根证书植入了现代浏览器，但是为了保证老客户端上证书的使用不受影响，他们还会把自己的根证书与另一个已有CA进行交叉签名。这种场景下，就不能简单地把你的证书“优化”成最短的路径，因为最短路径只能在最新的浏览器中工作，但在老客户端上证书链则是不完整的。

8.2.6　证书吊销

证书可以提供两种类型的吊销信息：CRL和OCSP。尽管不包含吊销信息的证书十分罕见，你还是应该检查一下自己的证书（比如使用SSL Labs检查网站，或者用OpenSSL的命令行工具）。

更为重要的是，你的CA需要提供一个可靠和快速的OCSP响应服务。因为每一次用户连接到你的网站，就会同样触发一次到CA站点的查询。为了获得最佳效果和可靠性，建议启用OCSP stapling特性。启用后，你的Web服务器可以直接返回OCSP响应，以避免因依赖CA网站而引起的性能、可用性和隐私方面的潜在问题。

8.2.7　选择合适的 CA

对于一个只需要DV证书的小网站，理论上所有CA都没问题。你可以随意选择，找一家最便宜的CA就行。任何公共CA都可以直接签发DV证书，并且无需你提供任何信息，为什么要付更多的钱呢？但是，如果你的证书需要用来保护重要的资产，花一点时间仔细选择，确保CA能满足你的所有需求。假如你还需要启用像钉扎这样的高级特性，选择一个CA将会是一个长期的承诺，请更加谨慎地选择。

- 服务

  今时今日，一切都是服务。如今的证书服务正变得越来越复杂。如果你的公司没有这方面的专家，也许可以找一个可靠的CA合作。成本虽然重要，但是用户接口和服务质量同样重要。

- 兼容性

  如果你有一个巨大的多样化的用户群，那么就需要一个被广泛预置的可信CA。老牌的CA厂商在这方面有明显的优势，因为他们早已花了大量时间将自己的证书预置到各种系统中。一个年轻的CA厂商，如果能与一个现有CA进行交叉签名，也一样没有问题。你最好这样做：(1) 列一张你需要支持的重要平台列表；(2) 找CA厂商要一份证书信任库部署列表；(3) 确保两者匹配。最后，申请一个测试证书在你的关键平台上进行测试验证。请记

住，不要只看CA现在支持的平台有哪些，还要看CA支持这些平台的时间点。有很多设备不支持更新信任库，因此时间点也是一个关键点。

- 先进性

  有些CA只对销售证书感兴趣，而有些CA却能塑造和引领整个行业。你应该与第二种CA合作而逐渐从第一种CA迁离。如今，务必选择这样的CA：已默认签发SHA256证书，提供良好的OCSP响应服务，并且有计划支持钉扎和证书透明度特性。

- 安全性

  作为一个CA，安全运行业务的能力显然是最重要的一个标准。但如何去判断CA的安全性？所有的CA都通过了审核，在表面上具有完全相同的安全性，但是历史证明不同的CA安全性上差异很大。最好的方法是仔细搜索一下这个CA安全性方面的历史。

自签名证书和私有CA

虽然本节一直在假设你从一个公共CA处获取证书，但其实你也可以决定是否使用自签名证书，可以创建自己的私有CA并用它给你的服务器签发证书。这三种方式都有自己的使用场景。

对于互联网网站，唯一安全的方法是使用公共CA来签发证书。

自签名证书是最差的方式。Firefox浏览器在自签名证书的使用上比较方便，你只要在第一次访问时创建一个例外，所有后续访问中的自签名证书都会自动被认为是有效的，其他浏览器可能每次都会弹出一个证书警告。[①]除非你每次对自签名证书进行校验，否则自签名证书的安全性是无法保证的，即使用的是Firefox浏览器也是一样。问问你自己：如果你的小组成员之前已经确认过一次自签名证书，当他们再次遇到证书警告时会怎么做？他们会与你确认证书是否更新，还是会直接放过？

理论上讲，使用私有CA总会是一个更好的方案。只需要多一点的前期工作，但一旦基础设施到位，根证书被安全地分发到所有用户设备上，接下来的使用就与其他系统一样，完全符合PKI体系的安全标准。

8.3　协议配置

在配置TLS协议时，你通常是根据安全性和互操作性的组合来作出选择的。在理想的环境中，只需要考虑安全性，因此你可以启用TLS 1.2并且禁用其他所有的版本。实际上，这种方案只能用于很少的场合或者严格可控的环境中。尽管现代浏览器都支持TLS 1.2，但其他许多产品和工具不一定支持。

① 通常可以绕过浏览器的用户界面，直接把自签名证书导入底层可信证书库。这样做的效果与创建Firefox例外是一样的，只是需要更多的工作，也更容易出错。

一个常见的公共网站通常支持TLS 1.0、TLS 1.1和TLS 1.2。SSL 2和SSL 3已基本废弃并且不安全。SSL 3确实被使用了很长的一段时间，但它有一个致命的缺陷，即2014年10月发现的POODLE攻击[①]。几乎所有客户端都至少支持TLS 1.0，唯一的问题是IE6浏览器，这个版本的IE在默认情况下只支持SSL 3。然而在2014年的POODLE攻击问题后，各大公司和CDN厂商都已经开始禁用SSL 3，这也会引导IE6用户升级到其他高级浏览器，实际上在IE6中可以手动启用TLS 1.0，但更推荐你直接升级到其他现代浏览器。

选择协议的另一个方式是遵守一个合适的安全标准。例如，2015年4月PCI委员会发布了PCI数据安全标准（PCI data security standard，PCI DSS）V3.1，[②]在这个版本的安全标准中，所有版本的SSL协议和TLS 1.0都被认为不够安全，不建议在新部署中使用，已在使用的系统可以保留到2016年7月再迁移到更安全的协议版本，推荐的协议版本是TLS 1.2。

> **注意**
> 旧版本的协议确实是一个隐患，因为大多数浏览器都可以被强制降级到所支持的最旧（最差）的协议版本，网络攻击者可以用这种方式禁用高版本协议并间接影响密码套件的选择。我们会在8.4节中详细讨论这个问题。

8.4　密码套件配置

在这一节中，我们会讨论影响密码套件配置的几个方面：加密强度、长期安全性、性能和互操作性。

8.4.1　服务器密码套件配置优先

无论你有什么样的客户端，强制服务器密码套件配置优先是确保最佳安全至关重要的一点。密码套件的选择是在TLS握手期间决定的，而TLS协议确保了握手的完整性，强制服务器算法优先后，网络攻击者就无法通过强制选择弱密码套件来发起攻击。

即使启用了服务器密码套件优先，你仍然不应该配置不安全的套件。网络攻击者可以强制一个浏览器（一般来讲其他客户端都不行，比如命令行工具）来自动将协议降级到最差的版本，如果你的服务器还在支持SSL 3，这意味着潜在的风险：加密没有身份验证，无法启用EC算法，有时甚至不能使用AES算法。

8.4.2　加密强度

务必使用128位以上强度的加密算法。尽管AES和CAMELLIA都符合这个条件，但AES有更

① This POODLE bites: exploiting the 3.0 fallback，http://googleonlinesecurity.blogspot.co.uk/2014/10/this-poodle-bites-exploiting-ssl-30.html（Google Online Security Blog，2014年10月14日）。

② PCI Council Publishes Revision to PCI Data Security Standard，https://www.pcisecuritystandards.org/pdfs/15_04_15%20PCI%20DSS%203%201%20Press%20Release.pdf（PCI Council，2015年4月15日）。

强的优势，因为它可以与GCM已验证套件一起使用。已验证套件是TLS提供的最好的特性之一，但请避免使用具有不安全性（虽然不一定是实际可利用的）的CBC套件。例如，美国国家安全局所定义的美国国家安全应用安全标准，即Suite B加密标准中，建议在TLS协议中只使用GCM套件。[①]

8.4.3　前向保密

建议不要使用RSA密钥交换，因为它不支持前向保密。RSA可以同时用于密钥交换和身份验证，不要搞混了，RSA用于身份验证是完全没有问题的，如果你使用的是RSA私钥，字符串RSA还是会出现在密码套件的名称中。推荐选择名称中包含字符串ECDHE或DHE的密码套件，从性能方面来说，ECDHE套件优于DHE套件。

使用前向保密后，用户的每次访问都会使用不同的密钥独立加密。如果没有前向保密，所有连接的安全性都完全依赖于服务器的密钥，一旦服务器密钥被破解或者泄露，所有历史通信都可以被直接解密，这是一个巨大的隐患，可以通过调整配置来直接修复。事实上前向保密特性是如此重要，未来的TLS版本预计将只会支持提供前向保密的套件。

对于ECDHE算法，secp256r1曲线模型可以为密钥交换提供128位的安全性。虽然现在还没有太多可供选择的曲线模型，但新的模型和使用机制正在被加入进来，例如在OpenSSL中会自动选择客户端支持的最佳模型。一旦这些新模型进入实用阶段，你应该尽量使用支持新模型的客户端。

对于DHE算法，你的DH参数配置应该提供2048位的安全性。新的服务器应用要么为所有连接提供这个加密强度，要么调整DH参数位数以匹配私钥的加密位数。然而在互联网平台上，尤其是一些老的平台，仍然在使用1024位甚至更低的加密强度。任何小于2048位的部署安全性都较弱，而任何小于1024位的部署都是不安全的。

将DH参数的强度增加到2048位在某些情况下会带来问题，例如Java 6客户端不支持大于1024位的DHE。如果你遇到了这种情况，有两个办法可以解决问题，也许都不是很理想，但根据实际情况可能有一种是可以接受的。

第一种方法，你可以继续使用弱DH参数（如果你的服务器不太可能成为政府机构的攻击目标）。弱DH参数可以被破解，但成本很高。如果你倾向于这一方案，则应当确保服务器不会使用常用的1024位DH组，换句话说，对每个服务器都生成唯一的DH参数。这种做法会大大提高攻击者的成本，你可以在6.5节中找到更多关于DH弱点的内容。

另一种方法，完全禁用DHE。这可能不像听起来那么糟糕，因为ECDHE密钥交换同样支持前向保密，并且比DHE更快。如今大多数客户端都已支持ECDHE，服务器在部署上也会更倾向于使用ECDHE而不是DHE。在最近观察的一个实际环境中，在一个流行的大众网站中只有百分之几的DHE流量。如果你保留使用弱DHE，意味着只有这一部分流量会被攻击（如果使用了自定义DH组，攻击成本会非常高）。如果你禁用DHE，这部分流量会失去前向保密但相对安全，除非有人通过攻击服务器获取了你的私钥。

① RFC 6460: Suite B Profile for TLS，http://tools.ietf.org/html/rfc6460（M. Salter和R. Housley，2012年1月）。

> **注意**
> 针对常用DH组的攻击只在低强度下有效，如果你的部署使用的是2048位密钥交换，就不用担心使用常用DH组（例如RFC 3526[①]所建议的）有什么问题。

8.4.4　性能

好消息是GCM套件是最快的，因此你无需在安全性和速度之间作出取舍。尽管AES和CAMELLIA在软件实现上加密速度相差不多，但是AES有一个优势，就是现代处理器的特殊指令集可以支持AES加速，因此AES在实践使用中要快得多。另外，硬件加速的AES能更好地防御缓存计时攻击（cache timing attack）。

请避免使用SHA256和SHA384的CBC套件用于完整性校验。CBC慢得多并且在安全性上不比SHA1强。不要把GCM套件名称中的SHA256和SHA384搞混了，已验证套件以不同的方式工作，并且性能也不慢。在完整性校验里也不要担心SHA1，配套使用HMAC散列算法时它是安全的。

对于ECDHE密钥交换算法，使用`secp256r1`曲线模型，它能提供128位安全强度并且有最好的性能。务必把ECDHE优先级设置成高于DHE，即使是设成常用的不太安全的1024位DHE也很慢；如果设置成2048位则更慢。

8.4.5　互操作性

互操作性的关键是提供尽量多的密码套件选择。TLS客户端是多种多样的，想必你也不愿无谓地丢弃掉一部分客户。如果你能按照此处的建议并设置好服务器算法优先，就可以让大多数客户端都在优选的密码套件上工作，而对于那些支持有限的老客户端则在降级的密码套件上工作。以下是一些老客户端的例子。

- 一些老客户端可能支持3DES和RC4，后者是不安全的且不应该使用，但是3DES可以提供112位的安全性，这仍然是可以接受的。
- 默认情况下，Java客户端不支持256位的套件。
- Java在版本8之前，DHE参数长度不能超过1024位。对于Java 7来说这不是什么问题，因为Java 7支持ECDHE，只要确保ECDHE的优先级高于DHE，就可以确保DHE不会被使用。如果你需要支持Java 6的客户端，就必须在没有前向保密的RSA密钥交换和支持前向保密的1024位DHE之间作出选择。
- 对于ECDHE密钥交换，只有两个曲线模型被广泛支持：`secp256r1`和`secp384r1`，如果你使用的是其他曲线模型，那么可能无法与一部分客户端（例如IE浏览器）协商成功ECDHE套件。

---

① RFC 3526: More MODP Diffie-Hellman groups for IKE，http://tools.ietf.org/html/rfc3526（T. Kivinen和M. Kojo，2003年5月）。

8.5　服务器配置和架构

保障系统整体安全性的唯一方法是确保系统的每个部分都是安全的。以下这些最佳实践总是适用的：禁用不必要的服务，定期安装系统补丁，以及做好严格的访问控制。此外，复杂的架构会带来自身的挑战，需要特别小心，最好是在设计阶段就能确保架构的变迁不会引入新的薄弱点。关于系统架构的全面介绍不属于本书的讨论范围，下面仅介绍与TLS部署有关的一些方面。

8.5.1　共享环境

共享环境在安全性上一般都有问题。共享主机不应该用于任何有商业加密需求的场合，有很多通过文件系统或直接内存访问的攻击可以导致私钥泄露。共享虚拟主机同样有类似的问题，是否使用取决于你的安全要求。数据加密有时候很容易被用错，尤其是资源被不相关的各方共享时。有些攻击方式依赖于对目标服务器的快速访问（例如，Lucky 13），而另外一些攻击方式（例如，缓存计时攻击）则依赖于对目标服务器上CPU的访问，虚拟主机方式给这些攻击都提供了可能性。

基础设施共享一般都是在成本、便利性和安全性之间的一种折中。众所周知，最好的安全环境需要专用硬件、强大的物理安全、完善的工程运营实践。

8.5.2　虚拟安全托管

一种被广泛接受的做法是为每个安全服务器配置一个独立的IP地址。主要的原因在于虚拟安全托管（将许多不相关的安全服务器置于相同IP地址）依赖服务器名称指示（server name indication，SNI）特性，而SNI特性在2006年才被加入到TLS中。因为出现得比较晚，许多老产品（例如，早期的Android版本、旧的嵌入设备，以及Windows XP上的IE浏览器）都不支持这一特性。因此，面向广大受众的网站应该继续使用这一做法，为每个站点配置独立的IP地址。

目前而言，是否依赖SNI的可用性实际处在一种模糊的边缘状态。如果你的站点用户都是新一代用户群，那就不需要顾虑这一点。我个人预见，在未来的几年内，将会看到大量只支持SNI的站点出现。考虑到对Windows XP的技术支持在2014年已经结束，XP用户应该会逐渐迁移到新的操作系统上。

8.5.3　会话缓存

会话缓存是性能优化的常见方案，客户端和服务器在首次建立连接并创建SSL会话时协商好传输密钥，在后续连接中就可以直接复用相同的传输密钥，大大减少CPU消耗和网络延迟。这个措施会导致安全性的降低，假如传输密钥被破解，整个会话的所有通信都可能被破解，但是一般情况下会话只会持续很有限的时间，所以这个折中方案被广泛应用于实际部署中。

我不建议禁用会话缓存，因为这会严重降低服务器的性能。即使是对安全性要求极高的网站，保持会话缓存在一天之内也都是可以接受的。如果你想要更高的安全性，可以缩短会话缓存的有效时间，例如一个小时。

当使用会话票证时，所有连接的安全性都依赖于同一个票证密钥。不幸的是，服务器的会话票证默认配置一般都有问题，大部分OpenSSL应用都直接使用了内置的默认票证密钥并且从来不会改变。当同一个票证密钥被使用几周或者几个月之后，前向保密实际也就基本失效了。因此当启用会话票证功能时，请务必配置一个新的票证密钥并且定期更新（例如每日）。以Twitter为例，它每12小时会更新一次密钥并在36小时后删除旧密钥。[①]

8.5.4　复杂体系结构

理想情况下，最安全的TLS部署是使用一个独立的服务器，并且明确定义好它的安全边界。现实情况中常见的是复杂体系结构，一般是分散在多个服务器上的多个组件和服务，这会带来新的问题和受攻击点。

- 分布式会话缓存

  如果一个站点的服务是由一个服务器集群来提供的，使用会话缓存来提高性能会比较困难。针对这个问题有两种常用方法：(1) 使用负载均衡的流量保持功能，保证统一客户端总是被分发到集群中的相同节点[②]；(2) 使用分布式会话缓存在所有节点之间共享TLS会话。会话缓存共享有一个安全上的隐患：攻击点会被放大，因为会话实际上会存在于多台服务器上。除此之外，在后端的会话缓存信息同步中往往会使用明文通信协议，意味着当攻击者渗透到后端网络时，可以很容易地窃取到所有传输密钥。

- 会话缓存共享

  在不同应用之间共享会话缓存会放大攻击面，因为它将一堆应用的安全性捆绑在一起，问题类似于前面讨论过的证书共享。建议若非必要，不要使用会话缓存共享（这可能会需要你额外的工作，因为不是所有服务器都允许缓存隔离），如果是使用会话票证方式，要确保每个不同的服务都使用不同的票证密钥。

- SSL卸载和反向代理

  SSL卸载是通过部署一个独立的接入层来负责加解密的工作。这个做法有其危险性，因为从接入层向后代理的通信是不加密的。尽管你可以认为内部网络通信是安全的，但这个架构设计确实引入了一个长期的严重攻击隐患：一旦攻击者入侵了内网，所有加解密就变得无效了。

- 网络流量监听

  RSA密钥交换设计时允许通过私钥共享来实现网络流量监听，通常可以通过入侵检测工具和网络监控工具来完成流量的自动解密。在一些特殊地方，监控所有网络流量可能是一个高优先级的目标。另一方面，这个功能与前向保密是有冲突的，因为所有通信的安全

① Forward Secrecy at Twitter，https://blog.twitter.com/2013/forward-secrecy-at-twitter（Jacob Hoffman-Andrews，2013年11月22日）。

② 通常这是基于源IP地址来实现的，有些负载均衡设备可以根据TLS会话ID来保证同一个用户始终分配到同一个服务器上。

都依赖这个共享的私钥，这将是一个长期的安全隐患。

- 基础设施外包

  外包基础架构的关键部分时，务必小心谨慎。基于云计算的部署越来越普及，但厂商往往不会提供关于服务实现的具体细节，这可能会导致不愉快的体验。2014年，一组研究人员分析了互联网CDN服务商的HTTPS使用情况，发现有一些居然在证书校验上就直接失败了。[①]

  最好的办法是自己做到对加密体系的完全控制。举个例子，如果你使用了Amazon的Elastic Load Balancer来实现应用的负载均衡，可以将它配置成纯TCP模式，然后完全在自己的服务器上做TLS加解密。

8.6 问题缓解方法

近年来，我们遇到过一系列的协议攻击以及各种TLS安全缺陷。有些很容易解决，比如打补丁，有些则需要仔细评估风险来制定合适的部署配置。在这一节中，我会讨论如何应对这些已知问题，但不会深入细节；你可以阅读第7章了解更多细节。

8.6.1 重新协商

不安全的重新协商是2009年的一个古老缺陷，但它仍然存在于大量的系统中。只要打上最新的补丁就可以修复。如果你没有使用客户端证书校验，可以在服务器直接禁用客户端重新协商功能，否则你应当升级支持安全的重新协商新标准。

还在使用不安全重新协商的服务器可能会遭到这些攻击：跨站点请求伪造（用户模拟）、信息泄露和跨站点脚本。这类攻击很简单并且有大量现成的工具。

8.6.2 BEAST（HTTP）

BEAST是2011年出现的，它利用块加密分组中可预见的初始化向量，针对TLS 1.0及更早版本中的CBC套件进行攻击。这种攻击实际上是浏览器的一个客户端问题（非交互型的客户端不受影响），所有浏览器的新版本都已经修复了这个问题，但使用老版本浏览器的用户（或者使用有缺陷的Java老版本的用户）还会有这个缺陷。虽然新的协议（从TLS 1.1起）对BEAST攻击有了很好的保护，但是老版本的浏览器又往往不支持新协议。BEAST攻击相对容易操作，可以用来窃取敏感信息片段（比如会话Cookie）。

8.6.3 CRIME（HTTP）

CRIME是2012年出现的，它是利用TLS和SPDY早期版本中压缩算法中的信息泄露的一种攻

---

① When HTTPS Meets CDN: A Case of Authentication in Delegated Service，http://cobweb.cs.uga.edu/~kangli/src/ieeesp2014.pdf（Liang等，IEEE Symposium on Security and Privacy，2014）。

击。与BEAST类似，CRIME也只能在浏览器中被利用，并且攻击目标也是获取请求头中的敏感信息片段（比如会话Cookie以及密码）。尽管大量的服务器支持TLS压缩，但只有很少的客户端支持，因此这种攻击的实际影响面较小。不管怎样，你都应该禁用TLS压缩功能，打上最新的补丁。

8.6.4　Lucky 13

Lucky 13是2013年出现的，它是针对CBC套件的一种攻击。它通过统计分析以及其他优化技术，利用块加密分组操作之间非常小的时间差来窃取信息，要做到这一点，攻击者需要非常靠近目标服务器。Lucky 13的攻击目标也是敏感信息片段，例如密码。

据我们所知，Lucky 13在现有流行的TLS库中都已经修复，修复方法是实现固定时间解密。确保在所有的环境中都部署上最新的补丁版本，就可以很好地避免这类攻击。尽管如此，CBC套件有其固有的弱点（即难以正确实现），未来可能还会有类似的问题，要达到最佳的安全性，可以改用TLS 1.2中的GCM套件来部署已验证加密。

8.6.5　RC4

历史上一直认为RC4是有安全弱点的，但从没有人认为它会影响到TLS加解密。这一点在2013年发生了改变，一些新的攻击方式得到披露，让我们知道它们可以利用RC4的弱点来恢复敏感信息的片段。2015年披露了更多的攻击方式，可以预见攻击者还会不断改进攻击方法。然而迄今为止所有已知漏洞都是在特定的条件下才能成立，在实际情况下并不具备可操作性。如上所述，你应该尽量避免使用RC4，除非确实有使用的需要。在某些时候，与其他攻击（比如BEAST、Lucky 13和POODLE攻击，后面会有详细说明）相比，选择RC4可能是两害择其轻。

截至2015年2月，RC4正式在TLS中被禁用[①]，然而这项禁令并没有对RC4的普及性产生多大影响。根据SSL Pulse在2015年3月的调查，RC4加密在测试中仍然被大约72%的服务器使用，虽然大部分情况是用来兼容老客户端，但也有22%的现代浏览器在访问时也会用到。

RC4与BEAST、POODLE以及Lucky 13的对比

RC4的使用不能孤立地讨论，我们必须考虑到上下文。RC4经常被用来防御那些更严重的问题，诸如BEAST、Lucky 13（Padding Oracle Attack）和POODLE等攻击手段都不能用于流式加密，而RC4恰好是唯一可用的流式加密。不幸的是RC4也有自己的弱点。那么应该怎么办呢？

现代浏览器通常不会受影响，因此你重点要考虑的是那些永远不会更新浏览器的老用户。BEAST攻击需要大量的投入，但是针对旧软件和老用户攻击仍然是可行的；POODLE更容易被攻击者使用，但也需要相当多的投入；Lucky 13的攻击已经被充分解决，只要你运行的是最新的服务器软件；另一方面，目前为止RC4的弱点被认为只能在可控环境下被利用，也就

① RFC 7465: Prohibiting RC4 Cipher Suites，https://tools.ietf.org/html/rfc7465（Andrei Popov，2015年2月）。

是说，针对RC4的攻击会越来越多，因为有其他问题的网站会越来越少。

对于大部分网站而言，最好的办法是着眼未来：部署最新（补丁）的TLS服务程序以预防Lucky 13，启用TLS 1.2以及GCM套件，禁用SSL 3和RC4。

对于一些高知名度的大型网站，还存在着大量使用有缺陷老浏览器的用户群，因此在你还无法禁用SSL 3时，可以考虑使用RC4来避免CBC缺陷攻击。当然在配置上仍然应当针对现代浏览器使用更强的加密协议（而非RC4!）；而对于TLS 1.0和更老的协议，请允许并强制使用RC4。

8.6.6　TIME 和 BREACH（HTTP）

TIME和BREACH是2013年出现的，它们基于CRIME的扩展，是针对HTTP压缩的攻击方式。与TLS压缩（从未广泛部署过）不同，HTTP压缩是非常有用且十分流行的，一般只会在有显著性能损失时才会被禁用。TIME攻击现在还只是一个理论上的概念，没有任何工具被披露。BREACH的发现者发布了源代码用于概念演示，意味着这种攻击更容易运用。这两种攻击都需要大量的投入来实现，这也使它们更适于针对具体目标的攻击，但无法普及。如果启用压缩，则BREACH可以用于窃取在HTML页面中任意位置出现的敏感信息片段。

解决BREACH缺陷需要付出更多努力，因为它的攻击面是在应用层。可以考虑以下两种解决方法。

- 隐藏敏感令牌

  对于敏感信息，例如CSRF防御或者会话管理的相关内容，最好的防御就是屏蔽这些内容。BREACH依赖在多个HTML页面中重复出现的字符串，一种有效的保护技术就是使屏蔽后的值每次都不相同但可以被程序正确解析。这种方法需要对应用代码进行相应的调整，不一定适用于遗留代码，但是很适用于在框架和库中实现。

- 引用信息不正确或不可用时禁用压缩

  禁用压缩可以避免攻击，但是代价太大。一般来说，攻击者总是从其他地方发起请求（不在你的网站上），这表示你可以检查引用信息，并仅当有可能受到攻击时才禁用压缩：当你发现请求来自其他网站或者当引用信息为空时（攻击者为了保护自己的隐私通常会加以隐藏）。这种方法不需要修改源代码，但是会有一定的性能损失，因为从其他网站过来的正常请求也都不会被压缩。

8.6.7　三次握手攻击

三次握手攻击是2014年出现的一种高效的攻击方式。它只能用于攻击那些启用客户端证书身份验证的系统。这种攻击的效果与不安全的重新协商类似但更容易被利用。短期来讲，最好的解决办法是使用最新版本的浏览器，它们都已经内置了反制措施。针对这些底层的核心问题，TLS协议本身也在不断改进中。

8.6.8　心脏出血

心脏出血是OpenSSL的一个漏洞，而OpenSSL是使用最广泛的SSL加密库。这个漏洞是2014年4月发现的，虽然这不是一个加密算法的问题，但是对服务器的影响是非常严重的。自从心脏出血漏洞被发现后，利用该漏洞的破解技术也迅速得到了开发，出现了大量现成的攻击工具，可以直接用来获取服务器的私钥信息。

解决这个问题需要几个步骤：(1) 给受影响的系统打上OpenSSL补丁，修复漏洞；(2) 更新私钥，重新签发证书，吊销老证书；(3) 如果使用了*会话票证*，更新票证密钥；(4) 评估可能存在于服务器内存中的敏感信息，并决定是否还要采取其他措施（例如一些网站建议用户更改密码）。

> **警告**
> 有些服务器虽然打了心脏出血补丁并且安装了新的证书，但没有更新私钥。这样做有很大的问题，因为在修复之前私钥有可能已经泄露，仍然可以被攻击者使用。

8.7　钉扎

互联网的信任机制完全依赖于数百家CA厂商颁发的证书，证书用来验证服务器的合法性。对于一般的网站来讲，这种方法很适用，因为这些网站不太可能被伪造证书；高知名度网站的风险则高得多，原因在于任意一个CA都可以签发任意一个域名的证书。为了避免这个问题，可以使用一种被称为*公钥钉扎*（public key pinning）的技术，它允许你强制指定签发证书的CA，只有被指定的CA为你的域名签发的证书才能正常使用。

钉扎技术大大降低了证书伪造的攻击可能性，但是也需要付出一些代价：需要一定的投入才能建立起一个成熟的钉扎战略和运营机制，并且钉扎完全依赖于浏览器的内部验证机制。关于钉扎技术有几个不同的标准，目前正处于不同的发展阶段：DANE（基于DNSSEC）、HTTP公钥钉扎和TACK。

8.8　HTTP

虽然SSL和TLS被设计成可以保护任意面向连接的协议，但最迫切的需求是用来保护HTTP网站。直到今天，网站加密仍然是TLS最常用的使用场景。经过多年的发展，互联网已经从一个简单的文件分发系统转变成一个复杂的应用交付平台，这种复杂性带来了更多的攻击可能性，也就需要我们在安全保护上投入更多。

8.8.1　充分利用加密

在HTTP交互中加密是可选的，导致的结果是许多网站实际都没有启用，即使业务上有需求也是这样。有些是设计上的问题，有些则是被简单忽略了。大部分未使用加密的情况，仅仅是因为它需要额外的付出和专业知识。有些人则认为加密会导致性能和成本方面的问题。另外，浏览

器允许安全和不安全的资源在同一个HTML页面中混合展示，这使得加密的使用更加混乱。

事实是，只要你将任何有价值的信息接入了网络，都必须加密，并且应该将整个站点完全加密！因为只做局部的加密几乎无法保障网站的安全性，尤其是在安全域和非安全域之间发生Cookie和用户的交互时，混合内容的问题（在一个安全页面中请求一个非安全的资源）同样可以被攻击者利用来侵入网站。

总之，最好的办法就是强制对整个域名进行加密，包括所有提供服务的子域名。

8.8.2 Cookie 安全

如果没有对HTTP Cookie做好安全措施（常见的程序错误），Cookie就很容易被网络攻击者获取，在极端情况下即便网站不以明文访问也是如此。因此，在QA阶段需要特别注意Cookie的创建。

此外，由于宽松的Cookie规范，攻击者很容易向毫无防备的应用中注入自己的Cookie。一般可以通过相关子域名下的其他应用来实现（比如，在blog.example.com中注入www.example.com），甚至可能是通过一个不存在的子域名来发起攻击，熟练的攻击者可以利用Cookie注入来提升自己的权限。为了达到最佳的安全性，要部署一个Cookie加密或者完整性校验的方案；前者更安全，但后者在JavaScript需要读取Cookie时更适用。

8.8.3 后端证书和域名验证

许多应用使用HTTPS进行后端通信，这种做法在系统程序、网站和移动应用中很常见。不幸的是，它们往往会犯同一个错误：没有正确地对证书进行校验，这样，系统实际上对攻击者敞开了怀抱。你的QA流程应该包括证书校验方面的测试。

大多数情况下，你只需要直接在底层TLS库中启用证书校验。有时候开发者依赖底层API提供的证书校验方法，但底层API不一定包含了协议的所有功能（例如域名检查），做为一个首要规矩，只要有更高级的替代方案，就应当避免使用底层API。

为了取得最佳的安全性，你可以考虑在应用中启用公钥钉扎。与浏览器不同，你无需等待钉扎标准的制定，在自己的程序中你对代码有完全控制，可以很容易地实现钉扎并大大降低被攻击的可能。

8.8.4 HTTP 严格传输安全

HTTP严格传输安全（HTTP strict transport security，HSTS）是一种标准，允许网站强制要求客户端使用加密方式，网站通过返回一个HTTP响应头来给浏览器指定策略。一旦服务器启用HSTS后，支持这一特性的浏览器将始终使用TLS与网站通信。它解决了一些其他手段难以解决的问题：(1) 用户保存了原始网站的书签，可直接访问明文网站地址；(2) 不安全的Cookie；(3) HTTPS stripping攻击；(4) 相同网站内的混合内容问题。

更重要的是，HSTS可以禁止浏览器使用无效证书。在没有启用HSTS时，浏览器在遇到无效

证书时会提示用户，但仍然允许用户继续访问，大多数用户无法区分这是攻击还是配置问题，因此往往会选择继续访问，从而导致很容易遭受网络攻击。当HSTS启用后，证书验证失败时用户将无法绕过，也就无法继续访问（其实TLS协议本来就是这样设计的）。

要达到最佳效果，HSTS应该在相关域名的整个名称空间内激活（例如exapmle.com和所有子域名）。

8.8.5　内容安全策略

内容安全策略（Content Security Policy，CSP）是一种机制，允许网站控制在HTML页面中嵌入的资源用什么协议来访问。与HSTS一样，网站通过返回一个HTTP响应头来给浏览器指定策略。尽管CSP最初是设计用来对抗XSS攻击的，但它实际上在网站加密方面起了更重要的作用：通过拒绝明文链接策略，可以禁用页面中的第三方混合内容。

8.8.6　协议降级保护

虽然TLS具有内置的协议降级保护，但有些浏览器在握手失败时会自愿降级，直接导致降级保护不起作用。这无疑是我们目前在协议实现上所遇到的最大的一个缺陷。

经过长时间的讨论，Google通过了一个特殊回退标记套件（fallback signaling suite）的提案，用来提示服务器潜在的降级攻击风险，并在Chrome中提供支持，Firefox从35版本开始支持[①]。这个机制的有效性还依赖服务器的支持。将这个特性加入到底层库后（OpenSSL从1.0.1版本开始支持），应用可以完全透明地工作。同时你也可以在应用外部实现这个机制，例如通过一个协议入侵检测系统。

① The POODLE Attack and the End of SSL 3.0，https://blog.mozilla.org/security/2014/10/14/the-poodle-attack-and-the-end-of-ssl-3-0/（Mozilla Security Blog，2014年10月14日）。

第 9 章

性能优化

人们有时会关心一下安全性，但始终对速度非常关心。没有人想让自己的网站速度更慢，提高性能的动机部分来自于我们对速度快的迷恋。例如，有很多证据表明，程序员都痴迷于性能，往往不愿意在代码质量上投入更多精力。另一方面，提高速度有利于增加收入是有据可查的。2006年，Google表示，其搜索结果每增加0.5秒会造成流量下降20%。[①]Amazon则表示，每增加100毫秒延迟会降低1%的收入。[②]

毫无疑问，TLS拥有变慢的“声誉”。主要是因为早期时候CPU还很慢，只有少数大的站点能买得起加密服务，但是今天计算能力不再是TLS的瓶颈。2010年，Google默认情况下在其电子邮件服务上启用了加密，之后他们表示SSL/TLS不再花费昂贵的计算成本：[③]

> 在我们的前端机器上，SSL/TLS计算只占CPU负载的不到1%，每个连接只占不到10 KB的内存，以及不到2%的网络开销。很多人以为SSL占用了大量的CPU时间，我们希望上述数字（首次公开）能消除人们的顾虑。

本章旨在尽可能接近Google的性能数据，很大一部分内容是关于减少延迟，大多数技术适用于任何协议（即使未使用加密），但对TLS尤其重要，因为它增加了连接开销。然后就是如何使用最少的CPU处理能力达到预期的安全性，并确保用户代理只需做尽可能最少的工作就可以验证证书。

> **注意**
> 在本章中，我着重TLS的性能调优，但在应用程序堆栈方面也有许多其他潜在的收益。对于更广泛的Web应用程序性能的话题，我推荐Ilya Grigorik的《Web性能权威指南》一书。（这本书的英文版于2013年由O'Reilly出版，提供免费在线阅读。[④]）

① Marissa Mayer at Web 2.0，http://glinden.blogspot.co.uk/2006/11/marissa-mayer-at-web-20.html（Greg Linden，2006年11月9日）。

② Make Data Useful，http://www.gduchamp.com/media/StanfordDataMining.2006-11-28.pdf（Greg Linden，2006年11月28日）。

③ Overclocking SSL，https://www.imperialviolet.org/2010/06/25/overclocking-ssl.html（Adam Langley，2010年6月25日）。

④ *High Performance Browser Networking*，http://chimera.labs.oreilly.com/books/1230000000545（Ilya Grigorik，检索于2014年7月17日）。

9.1　延迟和连接管理

网络通讯的速度由两个主要因素决定：带宽和延迟。[①]带宽用来衡量在单位时间内有多少数据可发送；延迟则描述一个消息从一端发送到另一端接收所需的时间。在这两者中，带宽是相对次要因素，因为通常你可以随时购买更多带宽；而延迟是无法避免的，因为它是在数据通过网络连接传输时被强加的限制。

延迟对于交互消息是一大限制因素。在典型的请求-响应协议中，请求到达其目的地并且等待响应返回需要一定的时间。这是我们测量延迟的措施，被称为*往返时间*（round-trip time，RTT）。

例如，每个TCP连接以*三次握手*开始：(1) 客户端发送一个`SYN`消息请求新的连接；(2) 服务器接受并以`SYN ACK`作出响应；(3) 客户端以`ACK`确认响应，并开始发送数据。它采用1.5次往返完成握手。在实践中，使用*客户端先行*（client-speaks-first）的协议，如HTTP和TLS协议，实际的延迟是一次往返，因为客户端可以在ACK信号之后立即发送数据（如图9-1所示）。

延迟对TLS影响特别大，因为它有自己精心设计的握手，在连接初始化的时候额外增加了两个往返。

9.1.1　TCP 优化

虽然有关TCP优化的完整讨论超出了本书的范围，但有两个非常重要并且易于使用的调优，每个人都应该了解。两者都关系到TCP内置的*拥塞控制*（congestion control）机制。在一个新的连接开始时，你不知道对端有多快。如果有足够的带宽，你可以用最快的速度传送数据，但如果你正在处理一个缓慢的移动网络连接呢？如果发送的数据太多，你会压跨连接，导致连接中断。出于这个原因，每一个TCP连接都有一个称为*拥塞窗口*（congestion window）的速度极限。这个窗口最初时较小，在可靠性能保证的情况下随时间增长。这种机制被称为*慢启动*（slow start）。

这给我们带来了丑陋的真相：所有的TCP连接，启动速度很慢，随着时间的推移速度增加，直到它们充分发挥其潜力。这对于HTTP连接往往是坏消息；它们几乎总是在不理想的条件下工作。

对于TLS连接，这种情况甚至更糟，TLS握手消息消耗了宝贵的初始连接字节（当拥塞窗口较小时）。如果拥塞窗口足够大，那么慢启动不会有额外的延迟。但是，如果较长的握手消息超过了拥塞窗口大小，发送方将必须把它拆分成两块，先发送一块，等待确认（一个往返），增加拥塞窗口，然后再发送剩下的部分。在本章稍后部分，我将讨论一些发生这种情况的示例。

1. 初始拥塞窗口调优

启动速度限制被称为*初始拥塞窗口*（initial congestion window，`initcwnd`）。如果要在现代平台上部署，则很可能已将初始拥塞窗口设置为一个较高的值。2013年4月发布的RFC6928[②]，建议

① What is Network Latency and Why Does It Matter?，http://www.o3bnetworks.com/wp-content/uploads/2015/02/white-paper_latency-matters.pdf（O3b Networks，检索于2014年5月11日）。

② RFC 6928: Increasing TCP's Initial Window，http://tools.ietf.org/html/rfc6928（Chu等，2013年4月）。

默认情况下初始拥塞窗口设置为10个网络段（约15 KB）。早期的建议是使用2~4个网络段起步。

在旧版本的Linux平台上，你可以改变所有路由的初始拥塞窗口：

```
# ip route | while read p; do ip route change $p initcwnd 10; done
```

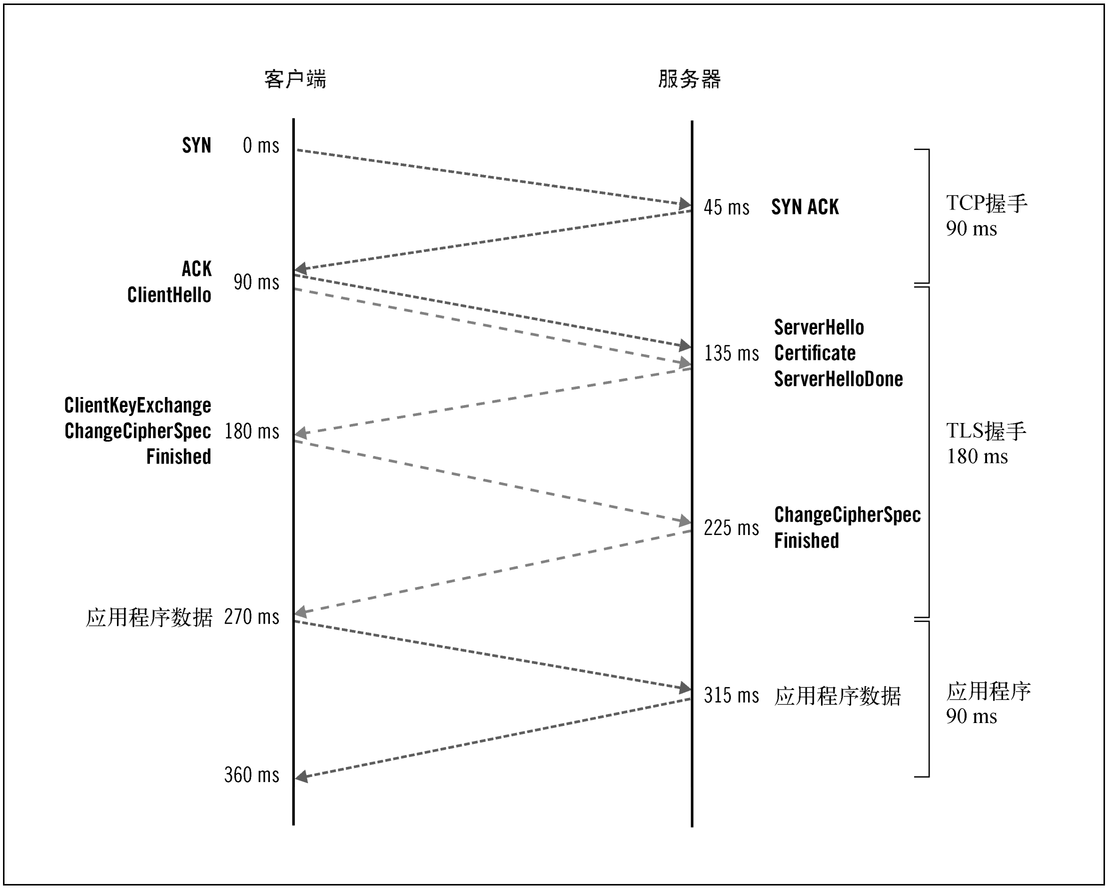


图9-1 TCP和TLS握手延迟

2. 防止空闲时慢启动

另一个问题是，慢启动可以作用于一段时间内没有任何流量的连接上，降低其速度，并且速度下降非常快。不活动的周期可以是非常小的，例如，一秒钟。这意味着，在默认情况下，几乎每一个长连接（例如，使用HTTP长连接）都有可能被从快调到慢！为了达到最佳效果，最好禁用这个功能。

在Linux上，可以在连接空闲时禁用慢启动：

```
# sysctl -w net.ipv4.tcp_slow_start_after_idle=0
```

可以通过将该设置添加到/etc/sysctl.conf配置使其永久生效。

9.1.2 长连接

大部分TLS性能影响集中在每一个连接的开始握手阶段。一个重要的优化技术是在连接数允许的情况下尽可能保持每个连接不断开。有了长连接，可以最小化TLS开销，同时也提高了TCP的性能。正如我们在9.1.1节所说的，保持TCP连接时间越长，传输越快。

在HTTP中，大多数事务往往是非常简短的，称作短连接。最初标准并没有提供一种方法让连接长时间保持，在HTTP/1.0中保持活动状态（keep-alive）是作为一个试验性功能出现的，直到在HTTP/1.1中才默认启用。

保持大量长连接是很有挑战性的，因为很多网站服务器的设计对这种情况处理得并不是很好。例如，Apache最初设计一个worker（进程或线程，这取决于配置）处理一个连接。这种方式的问题是，慢的客户端可以耗尽所有可用的worker并且阻塞Web服务器。此外，攻击者很容易创建大量的连接并且以很慢的速度发送数据来进行这种攻击。①

现在的趋势是使用事件驱动的Web服务器，通过使用固定的线程池（甚至单个执行线程）处理所有通讯，从而减少每个连接的成本以及被攻击的可能性。Nginx是一开始就内置以这种方式来操作网络服务器的例子。Apache在支持的平台上也开始默认使用事件驱动模型。

长连接的缺点是在最后一个HTTP连接完成之后，服务器在关闭连接之前会等待一定时间（保持活动状态超时，keep-alive timeout）。虽然一个连接不会消耗太多的资源，但是启用长连接降低了服务器的总体伸缩性。适用长连接的场景，最好是客户端突发大量的请求。最坏的情况是，当客户端只发送一个请求，并保持连接打开，但之后没再发送其他请求。

> **警告**
>
> 当配置较大的长连接超时时间时，限制并发连接数以免服务器超负荷是至关重要的。通过测试调整服务器，使其运行在容量限制内。如果TLS是由OpenSSL处理的，请确保服务器正确设置SSL_MODE_RELEASE_BUFFERS标志。②

很难推荐一个长连接超时的时间，因为不同的网站有不同的使用模式，60秒也许是一个不错的起点。每个站点可以在监控用户代理行为的基础上选择一个更好的时间值。③

长连接超时是有上限值的，无论服务器如何设置，用户代理都有自己的最大值。在我的测试中，在Windows 7上的Internet Explorer 11在30秒后关闭了连接，Safari 7是60秒，而Chrome 35是300秒。Firefox 30默认使用115秒为长连接超时时间（在`about:config`里面有个`network.http.keepalive.timeout`参数），除非服务器要求不同的值。Firefox乐于与服务器保持长连接，直到服务器关闭连接为止。

---

① Slowloris HTTP DoS，https://web.archive.org/web/20150315054838/http://ha.ckers.org/slowloris/（RSnake等，2009年6月17日）。

② SSL_CTX_set_mode(3)，https://www.openssl.org/docs/manmaster/ssl/SSL_CTX_set_mode.html（OpenSSL，检索于2014年7月6日）。

③ 这可以通过记录每个连接到Web服务器的保持活动状态访问日志来完成。在第13章和第16章中都说明了如何配置。

9.1.3 SPDY、HTTP 2.0 以及其他

启用TCP和HTTP长连接的收益只有这么多了。为了更大的提升，Google在2009年开始试验一个名为SPDY的新协议[①]。它在TCP层和HTTP层之间引入了一个新的协议层以提升速度。SPDY处在中间层，无需修改HTTP本身即可改善HTTP连接管理。

使用SPDY可以复用多个HTTP请求和响应，这意味着浏览器永远只需与每个服务器建立一个连接。单独的HTTP要实现相似的性能，浏览器必须使用多个并行连接。单个长连接提升了TCP利用率，降低了服务器的负载。

SPDY在各种情况下显示出性能改进，是一个巨大的成功。也许最重要的是，SPDY试验导致了全行业的努力，设计了同一概念的HTTP2.0[②]，唤醒了沉睡多年的HTTP协议：上一个版本是1999年发布的HTTP1.1。

虽然HTTP2.0仍处于开发阶段，但是SPDY可以部署使用。客户端支持对于新的浏览器来说都还不错：Chrome和Firefox都支持它有很长一段时间了，IE浏览器在2013年增加了支持（虽然只在11版在Windows 8.1上运行），而苹果公司宣布将在OS X Yosemite上支持SPDY。在服务器端，流行的网络服务平台（如Apache和Nginx）要么直接支持SPDY，要么可以进行扩展来支持SPDY。

我们可以预料到SPDY和HTTP2.0将挤出TCP更多性能，但接下来呢？一种选择是试图进一步提高TCP的性能。例如，TCP*快速打开*（Fast Open）是一种从TCP握手去除一个往返的优化技术。[③]另外，我们可以看看完全绕过TCP。另一项以Google为首的试验，叫作QUIC（quick UDP internet connection，快速UDP网络连接）[④]。它是建立在UDP之上的全新可靠连接协议，旨在同时提高性能（更好的连接管理、拥塞控制和丢包的处理）和安全性（默认情况下使用加密）。

9.1.4 内容分发网络

如果你维护一个面向全球用户的网站，就需要使用*内容分发网络*（content delivery network，CDN）来实现世界级的性能。一句话概括，CDN是利用地理上分散的服务器提供边缘缓存和流量优化（通常也称为*广域网优化*，WAN optimization）来产生价值的。

大多数时候，当你需要扩展一个网站时，投入大量金钱对问题会有所帮助。如果你的数据库负载重到快崩溃了，那么可以买一台更大的服务器。如果你的网站无法在一台服务器上运行，那么可以部署一个集群。然而，再多的钱也无法消除网络延迟。用户离你的服务器越远，访问网络就越慢。

在这种情况下，连接建立是一个很大的限制因素。TCP连接以三次握手开始，这需要一个往返来完成。然后是TLS握手，这需要两个额外的往返，因此HTTPS总共要三个往返。[⑤]对于距离

---

① SPDY，http://www.chromium.org/spdy/（The Chromium Projects，检索于2014年6月27日）。

② HTTP/2，https://en.wikipedia.org/wiki/HTTP_2（维基百科，检索于2014年6月27日）。

③ TCP Fast Open，https://en.wikipedia.org/wiki/TCP_Fast_Open（维基百科，检索于2014年6月27日）。

④ QUIC，https://en.wikipedia.org/wiki/QUIC（维基百科，检索于2014年6月27日）。

⑤ 同样的延迟适用于任何客户端先发起的协议。服务器先发起的协议延迟是2.5个往返，因为服务器可在其`Finished`消息后立即发送应用程序数据。

RTT大约30毫秒的周边用户，这大约需要消耗90毫秒，但对于距离很远的用户，这个建立连接的时间将会大得多。

为了服务器尽可能接近最终用户，CDN典型地经营着大量的地理分布的服务器。为了达到就近接入，CDN降低延迟通常有两种方式，即边缘缓存和连接管理。

- 边缘缓存

  由于CDN服务器贴近用户，可以将你的文件提供给用户，就像你的服务器真的在那里一样。有些CDN允许你推送文件给它们。这种方法提供了最好的控制和性能，但它更难以管理。其他的CDN作为反向代理运作（它们通过HTTP检索文件时，将需要的文件缓存在本地一段时间）；它们没有经过优化，部署比较随意。

- 连接管理

  缓存对于CDN部署是最好的，但并不适合所有的网站。如果你的内容是动态的、用户特定的，那么还是需要你的服务器提供实际的服务。但是，一个不错的CDN即使没有任何缓存，通过连接管理也会有所帮助。第一反应是这似乎违反直觉。通过CDN与直接到原始服务器相比会更快吗？答案是肯定的，CDN可以通过复用长时间保持的长连接消除大部分建立连接的成本。

  建立连接期间大部分的时间都花在等待上面。你发送一个数据包，并等待响应。当另一端在很远的时候，你会等待很久；但是，当另一端在附近的时候，你会得到一个快速的响应。为了尽量减少等待，CDN通过自己的基础设施将流量路由到距离目的地最近的一个点。因为是CDN自己完全可控的服务器，所以它可以保持内部连接很长一段时间。如果使用TCP协议，这意味着不存在连接建立，并且连接以最大速度运行。当然，为了更出色的性能，它们也可以使用专有协议和连接复用。

  当使用CDN时，用户连接到最近的CDN节点，这只有很短的距离。因为距离短，TLS握手的网络延迟也是很短的，例如，对于5毫秒的往返时间，握手只要15毫秒。一个新的TLS连接在理想的情况下，CDN可以复用现有的远距离长连接，从该节点一路去到最终目的地。这意味着不用其他进一步的工作；与CDN快速初始TLS握手后，用户与服务器的连接已经有效建立，应用程序数据可以开始传送。

  当然，并不是所有的CDN都在这样复杂的内部网络运行；一旦决定使用CDN，就有必要研究实施细则。更好的方法是测试CDN的实际性能，如图9-2所示。

> **注意**
>
> 并非所有CDN都是一样快的，特别是对于本章提及的TLS性能最佳实践。在你决定使用哪个CDN之前，一定要检查它们是否支持最快的TLS速度。Ilya Grigorik致力于提高TLS性能，并在他的网站维护了一个CDN对比图表。[1]

[1] CDN & PaaS performance，https://istlsfastyet.com/#cdn-paas（Is TLS Fast Yet?，检索于2014年6月27日）。

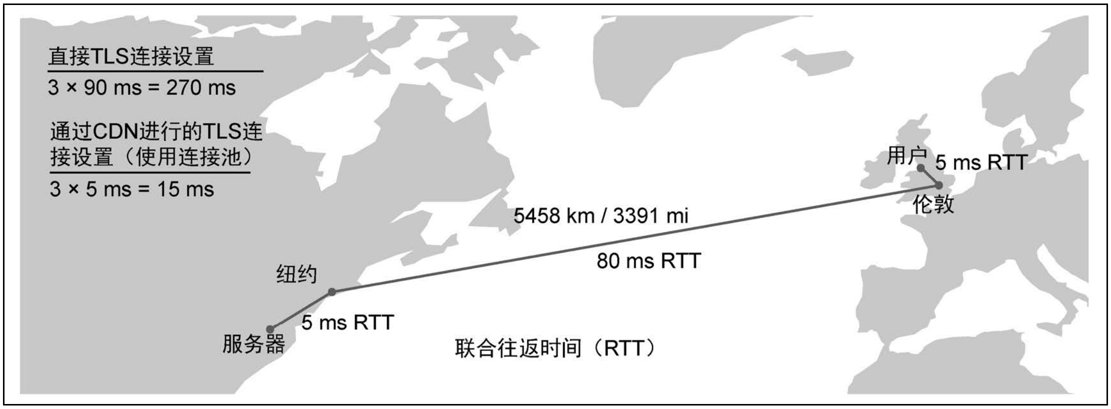


图9-2 对于通过CDN进行连接和直接连接这两种方式，TLS建立连接的耗时对比

9.2 TLS 协议优化

在连接管理之后我们开始专注于TLS的性能特征，以使你在理解影响TLS性能的每个方面之后，具备对TLS协议进行安全和速度调优的知识。

9.2.1 密钥交换

使用TLS最大的成本除了延迟以外，就是用于安全参数协商的CPU密集型加密操作。这部分通讯称为密钥交换（key exchange）。密钥交换的CPU消耗很大程度上取决于服务器选择的私钥算法、密钥长度和密钥交换算法。

- 密钥长度

  为了实现安全，密码学依赖进程对相关密钥的相对快速的访问，否则成本会非常高，也非常消耗时间。破解密钥的难度取决于密钥的长度，密钥越长越安全。但较长的密钥同时也意味需要花费更多时间进行加密和解密。为了达到最好的效果，选择刚好满足安全要求等级的密钥长度就好。

- 密钥算法

  目前你有两种私钥算法可以使用：RSA和ECDSA。[①]在过去很长一段时间里，RSA是唯一的选择，因此到现在RSA依旧是最主要的算法。当前RSA密钥算法推荐最小长度2048位，并且考虑到将来会更多部署3072位，RSA开始变得越来越慢。ECDSA会快很多，因此越来越有吸引力了。中等长度256位的ECDSA提供与3072位RSA一样的安全性，却有更好的性能。

① 虽然该协议包括许多DSA（DSS）套件，DSA密钥长度2048或者更高并没有得到广泛支持。最大长度值为1024位是不安全的。

- 密钥交换

  理论上你有三种密钥交换算法可以选择：RSA、DHE和ECDHE。RSA不提供前向保密，因此不推荐使用。在剩下的两个算法中，DHE太慢，你只能选择ECDHE。

  DHE和ECDHE密钥交换算法的性能取决于配置的协商参数长度。对于DHE，常用的是1024位和2048位的，分别提供80和112位安全等级。对于ECDHE，安全和性能取决一种称为曲线的东西。secp256r1曲线提供128位安全等级。另一个选择就是secp384r1曲线，比secp256r1曲线在服务器端要慢30%，带来的安全性提升却没有太大的意义。

你在实践中不能随意组合私钥和密钥交换算法，而是可以使用由协议指定的组合。一共有4种可能：RSA、DHE_RSA、ECDHE_RSA和ECDHE_ECDSA。为了更好理解这些套件的性能差异，我针对2048位RSA密钥和256位ECDSA密钥分别运行了4种套件的测试。这些会是你期望用于网站的密钥长度。DHE密钥交换代表两种DH参数长度：1024位和2048位。ECDHE密钥交换使用secp256r1曲线。

我使用配备Intel XeonE5-2670 2.5 GHZ处理器的Amazon EC2 m3.large实例进行测试。测试程序[①]是在Vincent Bernat的OpenSSL轻量级压测工具[②]基础上修改的。我测试了Ubuntu 14.04 LTS 附带的OpenSSL 1.0.1f。压测工具启动两个进程(一个客户端，一个服务器)，顺序执行1000次TLS握手并且在进程最后对CPU使用率进行测量。你会在图9-3中看到结论。

从测试结果我们可以得到以下结论。

- 使用RSA算法的服务器可以通过配置ECDHE密钥交换算法和ECDSA私钥来启用前向保密，并且提升握手性能。
- 启用前向保密（使用ECDHE密钥交换算法）同时保留RSA私钥身份验证会稍微降低握手性能，但对整体性能没有太大影响。
- DHE密钥交换即使是安全性较差的1024位参数也比较慢，并且安全性较高的2048位参数则会更慢。如果你在意性能，DHE应该作为最后的选择。因为新的客户端支持ECDHE，你可以给DHE套件设置较低的权重，以便只给老版本的客户端使用。Twitter数据显示75%的客户端使用ECDHE，[③]意味着高达25%的客户端可能使用较慢的DHE。
- DHE密钥交换算法取决于使用的参数长度，与ECDHE密钥交换相比，会让服务器端握手增加320到450字节。这是因为ECDHE密钥交换由名称引用标准化参数，但是DHE密钥交换需要服务器每次都选择协商参数发送给客户端。[④]
- 在使用ECDHE和ECDSA的时候客户端会有更多的开销，但这不是问题，因为客户端在某一时候只会发起少量连接。相反，服务器不得不并行处理成千上万的连接。

---

① ivanr/ssl-dos，https://github.com/ivanr/ssl-dos（GitHub，检索于2014年6月27日）。

② SSL/TLS & Perfect Forward Secrecy，http://vincent.bernat.im/en/blog/2011-ssl-perfect-forward-secrecy.html（Vincent Bernat，2011年11月28日）。

③ Forward Secrecy at Twitter，https://blog.twitter.com/2013/forward-secrecy-at-twitter（Jacob Hoffman-Andrews，2013年11月22日）。

④ 我在2.3节中论述了密钥交换消息的结构。

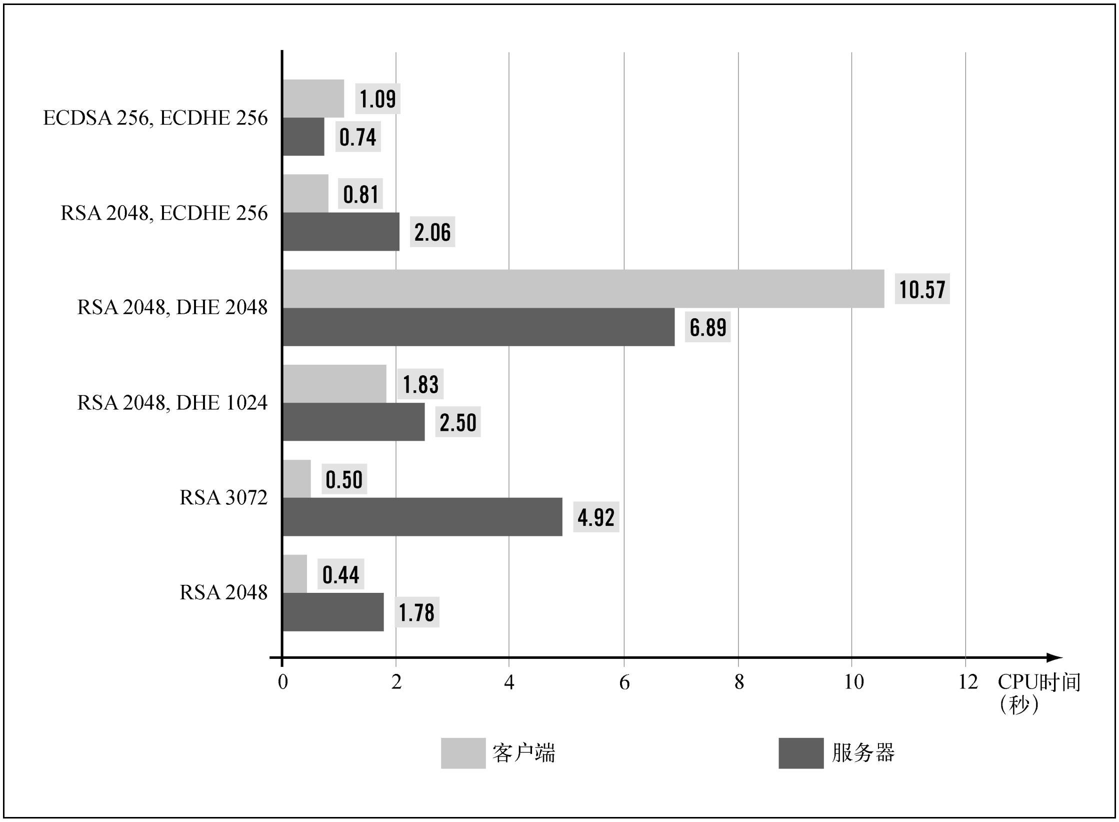


图9-3 TLS各种密钥交换算法性能对比（条形越短性能越好）

> **注意**
> 这里呈现的测试结果只作参考，客户端和服务器都使用特定的OpenSSL版本。实际上不同版本的TLS对应不同的依赖库、设备和CPU，会有显著的性能差异。

我建议查阅Huang等关于前向保密部署的性能研究报告，以进一步了解更详细的密钥交换性能。[①]另外Symantec在2013年白皮书里关于椭圆曲线加密算法性能的讨论也是一份很好的参考资料。[②]

① An Experimental Study of TLS Forward Secrecy Deployments，https://www.linshunghuang.com/papers/ecc-pfs.pdf（Huang等，2014年）。

② Elliptic Curve Cryptography (ECC) Certificates Performance Analysis，http://www.symantec.com/content/en/us/enterprise/white_papers/b-wp_ecc.pdf（Kumar等，2013年6月12日）。

False Start

Google在2010年提议修改TLS协议，将完整握手的延迟从两次往返减少到一次。[①]通常情况下，一次完整的TLS握手需要两次往返，包含4个协议消息碎片（客户端和服务器各2个），TLS只允许在完整握手完成之后才开始发送加密的应用数据。False Start建议调整协议消息的时间，假设握手会成功，在完整握手完成之前我们就可以开始发送应用数据。

False Start这个改变让性能提升有了可能。Google举证False Start可以减少30%的握手延迟，这是很了不起的。[②]这个改变通常是没问题的；副作用是，如果受到攻击，客户端会将加密数据发送给攻击者。此外，完整的握手完成之后才可以进行完整性验证，用于加密的参数有可能已经被攻击者修改。

为了应对这种攻击向量，Google建议只对足够安全的加密使用False Start：私钥足够强大、支持前向保密密钥交换算法和128位安全等级的密码套件。在不需要强密钥交换时，这会留下漏洞。在之后的2015年5月，一个称为Logjam的攻击（6.5节中对该攻击进行了介绍）表明，主动的攻击者有可能迫使支持False Start的浏览器发送低至512位安全性保护的HTTP请求。

尽管有性能提升，Google还是在2012年宣布False Start失败了，因为互联网上有太多不兼容的服务器。[③]不过Google并没有完全停止它，Chrome浏览器在服务器端实现NPN扩展的情况下（用于协商SPDY协议，现已被ALPN取代）继续使用False Start，这被认为是安全的。其他浏览器遵循和采用了类似的行为。Firefox浏览器从28版本开始支持False Start[④]，触发条件与Chrome相同。Apple在OS X 10.9中增加了这个支持，条件是强大的密码套件和前向保密，但不需要NPN。[⑤]IE浏览器从10版本开始支持，按照原来的建议实现False Start，对不支持的网站加入黑名单列表以禁用。[⑥]

False Start对于支持前向保密是很大的激励。使用False Start不仅安全性明显更好，而且性能也得到了提升。

9.2.2 证书

一次完整的TLS握手期间，服务器会把它的证书链发送给客户端验证。证书链的长度和正确

① TLS False Start，https://tools.ietf.org/html/draft-bmoeller-tls-falsestart-00（Langley等，2010年6月）。

② SSL False Start Performance Results，http://blog.chromium.org/2011/05/ssl-falsestart-performance-results.html（Mike Belshe，The Chromium Blog，2011年5月18日）。

③ False Start's Failure，https://www.imperialviolet.org/2012/04/11/falsestart.html（Adam Langley，2012年4月11日）。

④ Re-enable TLS False Start，https://bugzilla.mozilla.org/show_bug.cgi?id=942729（Bugzilla@Mozilla，bug #942729）。

⑤ `sslTransport.c`，http://opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslTransport.c（Apple Secure Transport源代码，检索于2014年5月5日）。

⑥ Networking Improvements in IE10 and Windows 8，http://blogs.msdn.com/b/ieinternals/archive/2012/08/01/internet-explorer-10-network-performance-improvements-first-available-pre-resolve-pre-connect-caching.aspx（Eric Lawrence，IEInternals，2012年8月1日）。

性对握手的性能有很大影响。

- ❑ 使用尽可能少的证书

  证书链里的每个证书都会增大握手数据包。正如前面讨论过的那样，证书链包含太多证书有可能会导致TCP初始拥塞窗口溢出。在SSL早期的时候，CA会直接使用根证书进行服务器证书签发，但这是非常危险的（根证书私钥应该离线保存），已经逐步被弃用。现在你的证书链里面最好有两个证书：一个服务器证书和一个签发CA的证书。

  证书链长度并不是唯一的要素；证书链中的每个证书必须验证签名与签发证书里的公钥匹配。用户代理决定是否还要对每个证书的吊销状态进行检查。

  虽然我不会建议你基于证书信任链的长度选择CA，但你还是需要提前检查一下证书链是否太长。

- ❑ 只包含必需的证书

  证书链里面包含非必需的证书是一个常见错误。每个这样的证书会给握手消息额外增加1~2 KB。

  通常根证书也会被包含在里面，虽然并无用处。用户代理要么信任根证书（已经内置有一份根证书的备份），要么不信任。证书链里面是否包含根证书并无区别。这是一个很普遍的问题，因为甚至一些CA的安装说明里面也把他们的根证书包含进去了。

  其他情况可能是因为配置错误导致的非必需证书。有种情况并不常见，服务器证书链包含之前证书遗留下来的中间证书。在某些罕见的情况下，服务器发送的证书链里面包含数百个证书。

- ❑ 提供完整的证书链

  对于可信任的TLS连接，服务器必须提供一个被根证书信任的完整证书链。另一种常见错误是提供不完整的证书链。尽管有些用户代理可以取得缺失的证书，但这些会涉及额外的HTTP查询，可能需要花费数秒时间。确保证书链有效才能达到最佳效果。

- ❑ 使用椭圆曲线证书链

  因为ECDSA私钥长度使用更少的位，所以ECDSA证书会更小。Huang等（2014年）观察到256位的ECDSA证书链比2048位的RSA证书链大约要小1 KB。

- ❑ 小心同一张证书绑定过多域名

  近来几十个网站共享一张证书是比较普遍的，在某些情况下甚至还有数百个网站共享一张证书的。这样允许多个网站共享同一个IP地址，以便支持那些不支持虚拟安全网站［通过服务器名称扩展（server name extension）或者SNI］的客户端访问。每增加一个域名都会增加证书的大小。少量域名不会有明显的影响，但是数百个域名就有可能。

  如果你想多个网站在同一个IP地址上又同时保持最小握手长度，有个技巧：(1) 对于支持SNI的客户端，在Web服务器上为每个域名配置单独的证书。(2) 取得一份你想共享相同IP地址的全部域名的后备证书，在Web服务器上配置以便支持那些不支持SNI的客户端。如果你这么做，支持SNI的客户端（占主要部分）可以取到它们要访问网站的独立证书，其他客户端（少数遗留的）会取得一个较大的多域名证书。

9

> **警告**
> 当需要对客户端进行身份验证时，你可以配置服务器通告，只接受哪些CA签发给客户端的证书。每一个这样的CA等同于它的可分辨名称。如果配置了太多CA，这个列表运行时可能有几KB，会降低性能。出于性能原因，你可以避免配置通告可接受的CA，因为它是可选的。

9.2.3　吊销检查

虽然证书吊销状态在不断变化，并且用户代理对证书吊销的行为差异很大，但作为服务器，要做的就是尽可能快地传递吊销信息。实际操作中转化为以下这些规则。

- 使用带OCSP信息的证书

  OCSP被设计用于提供实时查询，允许用户代理只请求访问网站的吊销信息。因此查询简短而快速（一个HTTP请求）。相比之下CRL是一个包含大量被吊销证书的列表。一些用户代理只有当OCSP信息不可用的时候才下载CRL，这种情况下浏览器与你的网站的通信将被暂停，直到CRL下载完成。几十秒的延迟都不少见，尤其是通过慢速的网络连接时（想想移动设备）。

- 使用具有快速且可靠的OCSP响应程序的CA

  不同CA之间的OCSP响应程序性能也有所不同。缓慢和错误的OCSP响应程序会潜在地导致性能下降，这个现实被隐藏了很长一段时间。在你向CA提交之前先检查他们的历史OCSP响应程序。更多信息参考5.10.4节的“响应程序可用性和性能”部分。

  另一个选择CA的标准是它更新OCSP响应程序的速度。为了避免网站错误，你希望自己的证书一被颁发就加入到OCSP响应程序中。令人费解的是，有些CA对于新证书的OCSP更新拖延很久，这个期间OCSP响应程序都会返回错误。

- 部署OCSP stapling

  OCSP stapling是一种允许在TLS握手中包含吊销信息（整个OCSP响应）的协议功能。启用OCSP stapling之后，通过给予用户代理进行吊销检查的全部信息以带来更好的性能。OCSP stapling增加握手大约450字节，会让握手略微变慢，但可以省去用户代理通过独立的连接获取CA的OCSP响应程序来查询吊销信息。

  OCSP响应大小因签发CA的实际部署不同而不同。与最终实体证书（正在检查吊销的）具有相同CA签名的OCSP响应会比较简短。因为用户代理已经有签发CA的证书，OCSP响应可以只包含吊销状态和签名。

  一些CA喜欢使用不同的证书对OCSP响应进行签名。因为用户代理事先并不知道其他证书，CA必须将它们包含在每个OCSP响应里面。这会给OCSP响应略微增加超过1 KB的大小。

> **注意**
> 当浏览器跳过吊销状态检查时，虽然会获得更好的性能，但也面临安全风险。EV证书始终检查吊销状态以提供更好的安全性。DV证书不总是检查吊销状态，可能有轻

> 微的性能优势。这个问题可以通过OCSP stapling让两种类型的证书都有相同的性能来解决。

9.2.4 会话恢复

TLS理解两种类型的握手：完整握手和简短握手。理论上完整握手只会在客户端与服务器建立TLS会话（TLS session）的时候进行一次。后续的连接，双方使用简短握手恢复之前协商的会话。简短握手因为不需要花费昂贵的加密操作会更快，并且少一次往返时间。一个良好的恢复率可以降低服务器负载并且缩短最终用户的延迟。

TLS会话恢复由通讯的双方共同控制。你这边的目标应该是配置会话缓存，让各个会话有效期在一天左右。除此之外，剩下就是由客户端决定何时恢复会话，何时开始一个新的会话。我的个人经验以及其他证据表明，你可以期待合理配置的服务器有50%的会话恢复率。

9.2.5 传输开销

TLS协议的最小传输单位是一个TLS记录，它最多可以包含16 384字节的数据。TLS记录在不加密的情况下做的事情不多，只有一点小的开销；每个记录以5字节的元数据开头：内容类型（1字节）、协议版本（2字节）和数据长度（2字节）。流加密、分组加密和已验证密码套件的TLS记录开销如图9-4所示。

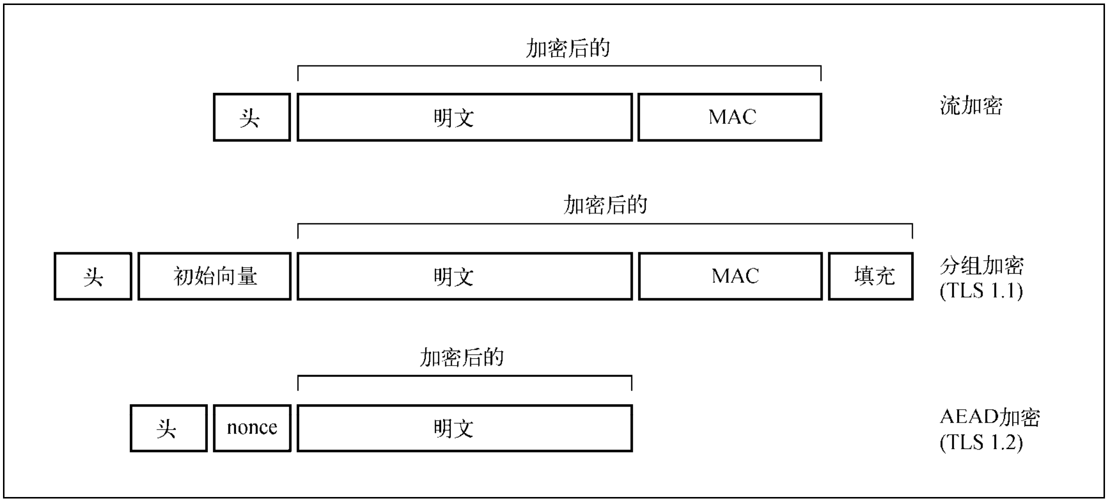


图9-4 流加密、分组加密和已验证密码套件的TLS记录开销

加密和数据完整性算法引入额外的开销，因协商时不同的密码套件而不同。流加密产生较小的开销，因为它是输入一字节输出一字节；开销只来源于完整性校验。

分组密码有更多的开销，因为每个TLS记录需要增加一个与加密块大小一样的显式向量，并

且填充使得明文长度正好变成若干个加密块的大小。填充的长度由数据长度决定，但平均有二分之一的块大小。当前最安全加密算法都设计了16字节的块大小。

自带完整性验证的加密算法的传输开销处于中间位置：它们不使用填充，但每个记录包含了一个8字节的随机数。

表9-1显示了最常用密码套件的开销。

表9-1　各种广泛使用的加密算法传输开销

加密算法	TLS记录	初始向量/Nonce	填充（平均/最大）	HMAC/Tag	总计（平均）
AES-128-CBC-SHA	5	16	8/16	20	49
AES-128-CBC-SHA256	5	16	8/16	32	61
AES-128-GCM-SHA256	5	8	–	16	29
AES-256-CBC-SHA	5	16	8/16	20	49
AES-256-CBC-SHA256	5	16	8/16	32	61
AES-256-GCM-SHA384	5	8	–	16	29
CAMELLIA-128-CBC	5	16	8/16	20	49
3DES-EDE-CBC-SHA	5	8	4/8	20	37
RC4-128-SHA	5	–	–	20	25
SEED-CBC-SHA	5	16	8/16	20	49

正如你所见，不同密码套件之间的开销差异很大。最坏的情况是，使用AES和SHA256的套件平均增加61字节开销。最好的情况是，已验证模式的密码套件只增加29字节。这些开销和下一层的开销相比并不算大；TCP/IP开销是每个IPv4包52字节，每个IPv6包72字节。每个IP包的大小大约在1500字节，但是每个TLS记录可以大到16 384字节，很明显TCP比TLS造成更多的开销。

无论哪种方式，如果可能，你应该避免发送小包数据。缓冲应用层数据以确保更小的网络开销是必要的，除非需要实时传递短消息。例如动态构建一个HTML页面的时候，最好使用小的输出缓冲，4 KB足以让微小的数据包合并成一个大的数据包发送。我见过一些错误配置的应用，让每个数据写入（仅仅是很少的字节）产生一个TCP包，造成了巨大的网络开销。这类问题在直接使用套接字而不是基于Web应用程序框架、基础库实现的时候更加普遍。

如果你对自己的应用程序行为不确定（这并不罕见，现在我们的软件一般都有多层抽象），可以在网络层捕获流量以观察TCP包和TLS记录大小。

9.2.6　对称加密

关于CPU消耗，一旦TLS握手完成，就代表最坏的情况已经过去了。用于对称加密的加密操作有明显的CPU成本，它由加密算法、加密模式和完整性校验功能决定。

为了确定各种密码套件的性能特征，我用与本章前面相同的环境进行了进一步的测试。我明确选择一个支持AES-NI指令集的处理器，为AES加密提供硬件加速。[①]希望在意性能的网站使用

① 如果你要购买硬件，请检查CPU的规格来确定对AES-NI的支持。在云环境中，你同样能够通过检查供应商的文档来确定。在运行中的Linux服务器上可以在/proc/cpuinfo中查找aes标志。

类似的硬件进行操作。每个测试运行由两个线程组成（一个用于客户端，另一个用于服务器），向另一端发送大约1 GB数据，每次发16 KB。我测试了当前全部实用并且安全的密码套件，同时也有一些过时的套件作对比，如图9-5所示。

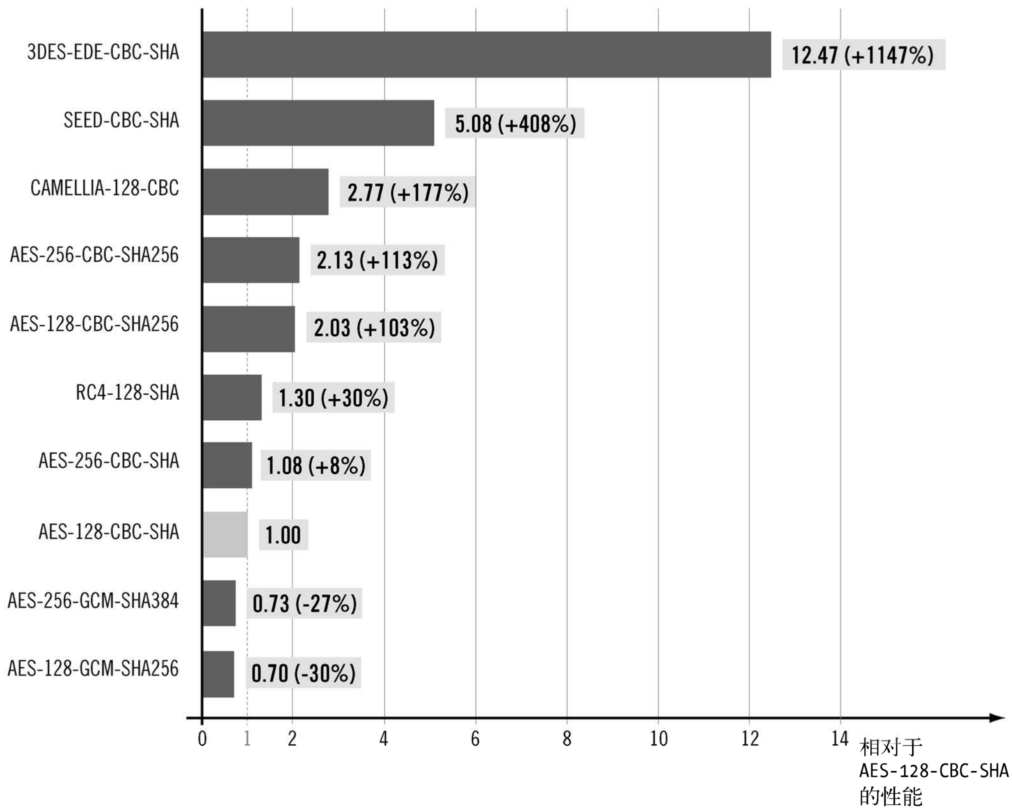


图9-5　各种密码套件性能基于AES-128-CBC-SHA的对比（越短性能越好）

我决定把AES-128-CBC作为参考基准，因为它在依旧安全的密码套件里面是最普遍被使用的。研究结果非常有意思。

- AES很明显是性能冠军。即使没有硬件加速，AES还是比除了RC4之外的其他加密算法要快。使用硬件加速的AES-128-CBC比CAMELLIA-128-CBC要快2.77倍。CAMELLIA-128-CBC与最快的AES算法AES-128-GCM-SHA256相比要慢4倍。
- 在TLS 1.2中定义的AES和SHA256相当慢，因为SHA256比SHA要慢很多。
- 在已验证（authenticated，GCM）模式下的AES-128比AES-128-CBC要快1.4倍，甚至比以前的性能冠军RC4-128-SHA还要快。这是非常鼓舞人心的，同时AES-128已验证模式也是当

前可用的最安全的算法之一。

- 旧的3DES和SEED套件要慢好多倍，应该避免使用。RC4虽然相当快但不安全，也应该避免使用。

虽然我们倾向于投入大量时间对服务器进行压测，但客户端的性能还是要留意的。较新版本的笔记本可能已经支持硬件加速AES，但是还有大量动力不足的移动设备没有支持。出于这个原因，Google当前试验了一个名为ChaCha20-Poly1305的新已验证密码套件。[①]虽然它只有硬件加速的AES的一半速度，但在移动设备上有3倍的性能提升，并且后续还有提升的空间。Google已经广泛使用这个新的加密算法，但我们还必须等标准化进程完成。[②]

9.2.7　TLS 记录缓存延迟

回顾前面的讨论，TLS记录是TLS发送和接收数据的最小单位。TLS记录的大小与下一层TCP包的大小并不匹配，一个全尺寸的TLS记录16 KB需要被拆分成许多小的TCP包，通常每个小于1.5 KB（如图9-6所示）。

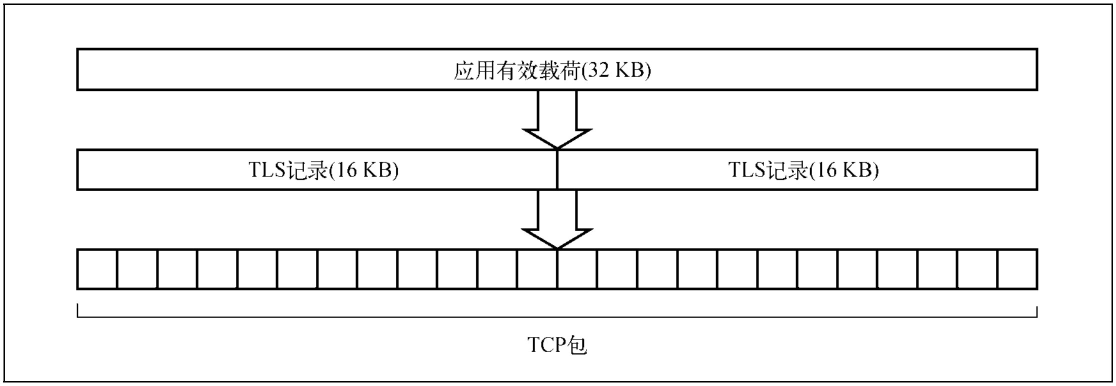


图9-6　使用TLS和TCP传输32 KB的应用程序数据的碎片示例

有一点需要注意：虽然整个TLS记录的各个碎片陆续会到达，但在全部到齐之前是无法进行处理的。这是因为TLS记录同样是数据解密和完整性检验的最小单位。这个缓存有时可能会导致延迟增加。

- 丢包和延迟

  虽然TCP可以把丢失和延迟的数据包恢复，但这仍然需要消耗一次往返。每一次额外的往返对于整个TLS记录都意味着延迟，不仅仅是丢包。

① TLS Symmetric Crypto，https://www.imperialviolet.org/2014/02/27/tlssymmetriccrypto.html（Adam Langley，2014年2月27日）。

② The ChaCha20-Poly1305 AEAD Cipher for TLS，https://datatracker.ietf.org/doc/draft-ietf-tls-chacha20-poly1305/（Langley等，2015年6月）。

- 初始拥塞窗口

  另一个触发额外往返的延迟是在连接初期发送大量数据导致初始拥塞窗口溢出。一旦拥塞窗口满了，发送端必须等待响应（一次往返），拥塞窗口增加再发送更多数据。

如果你的Web服务器支持TLS记录调整，就应该考虑将默认值（很可能是16 KB这么大的数值）改成更为合理的值。找到最佳大小需要一些试验，因为正如之前所讨论的，这由部署的密码套件和相应的传输开销决定。

如果你不想在这上面花太多时间，考虑使用4 KB这个合理的默认值。如果你想将TLS记录大小设置为与TCP包准确匹配，就从1400字节开始，然后通过观察数据包逐步调整。例如，假设IP层最大传输单元（maximum transfer unit，MTU）是1500字节：

```
  1,500 bytes MTU
-    40 bytes IPv6 header
-    32 bytes TCP header
-    49 bytes TLS record
-------------------------
= 1,378 bytes
```

静态的TLS记录大小无论使用什么值都有一些问题。首先MTU的值是变化的。虽然多数客户端继承以太网1500字节的限制，但也有一些协议支持更大的数据。比如所谓的巨型帧（jumbo frame）允许多达9000字节。其次，很容易错误计算并且指定一个错误的大小。例如，使用IPv4（头中20字节，而不是40字节）计算会略有不同，密码套件配置有变化也都有可能让这个数值不对。

另一个问题是减小TLS记录的大小会增加传输开销。传输16 KB数据使用大的TLS记录可能只增加50字节（0.3%）的开销。但如果你分成若干个记录，例如10个记录，开销变成了500字节（3%）。

可能最佳方案是由Web服务器来调整TLS记录大小，有两个原因：(1) 它可以在连接的开始就发现MTU；(2) 它可以在连接的生命周期里动态改变记录大小，当早期拥塞窗口较小的时候使用较小的值，而传输数据越多时就调得越大。HAProxy正是这么做的。[①]

9.2.8 互操作性

互操作性问题有时真的会影响性能，它会一直隐藏直到你知道如何准确发现它。例如，你的服务器与一些新版本协议的特性（例如TLS 1.2）不兼容，浏览器可能需要通过与服务器进行多次尝试，最终才能协商一个加密的连接。[②]但是除非你经历过这个问题并且意识到性能下降，否则你不会知道；服务器不能检测到它并且浏览器也不会警告你这一点。确保良好的TLS性能的最好方式是升级最新的TLS协议栈以支持较新的协议版本和扩展。

9.2.9 硬件加速

早期的SSL，公钥加密对当时可用的硬件来说是非常慢的。结果取得不俗性能的唯一办法就

① OPTIM: ssl: implement dynamic record size adjustment，http://www.haproxy.org/git?p=haproxy.git;a=commit;h=7bed945be0cc9c91cd0114a9bcb965702f9389e3（Willy Tarreau，2014年2月2日）。

② 多重连接尝试是现代浏览器所采用的自愿协议降级机制的一部分。我在6.6.3节中详细论述过。

是使用硬件加速。随着时间推移，普通CPU速度不断提高，加速设备开始失去了市场。[1]

运行着全球最大网站的公司很乐意使用软件处理加密。例如，下面是Facebook关于硬件加速的说法：[2]

> 我们发现当前基于软件的TLS实现在普通的CPU上已经运行得足够快，无需借助专门的加密硬件就能处理大量的HTTPS请求。我们使用运行于普通硬件上的软件提供全部的HTTPS服务。

今天人们购买硬件加密设备更多是因为它安全保存私钥的能力，很少是为了加速公钥加密。这种产品被称作硬件安全模块（hardware security module，HSM）。不过使用HSM可能会是你架构的一个瓶颈，因为这种设备很难大规模化。

取决于你的使用场景，硬件加速也可能是正确的。例如，你已经存在一个容量边缘的系统，相比其他硬件和架构改变，安装加速卡或许是更好的选择。

9.3　拒绝服务攻击

拒绝服务（Denial of Service，DoS）攻击在互联网上很常见，出于好玩或者利益。发起攻击很容易并且成本很低。另一方面，防御拒绝服务攻击却是费钱费时。任何一个小网站都很容易被任何想要尝试这种攻击的人压跨。对于大网站，如果他们可以挺住，只是因为他们花费大量金钱用于防御，并且攻击者没有尽力。

发起拒绝服务攻击最主要的方式是使用僵尸网络，一种由大量被感染的计算机组成的网络。服务器被当作僵尸节点因为它们有充足的带宽。家庭电脑虽然单机能力有限，但它们的价值是数量巨大。

如果有人想通过僵尸网络攻击你，TLS配置可能无所作为。不管是否使用TLS，坚定的攻击者可以持续增加僵尸网络的规模直到成功，这对攻击者来说开销并不大。这就是说，存在这样一个建议：扩展TLS以要求客户端在消耗服务器资源之前执行某些计算。[3]但是，防御拒绝服务攻击最终还是在于网络层。

- 连接限制

  这是“入门级”的拒绝服务防御措施，你可以在整个网络的入口部署专用设备，甚至为每台服务器的内核配置连接限制。[4]通过这种方法，你可以对抗一些简单的攻击，例如，来自少量IP地址的攻击。连接限制对于来自大量独立主机的流量泛洪攻击并没有太大帮助。

- 资源超配

---

① High Scalability for SSL and Apache，http://www.awe.com/mark/talks/apachecon2000.pdf（Cox和Thorpe，2000年7月）。

② HTTP2 Expression of Interest，http://lists.w3.org/Archives/Public/ietf-http-wg/2012JulSep/0251.html（Doug Beaver，HTTP工作组邮件列表，2012年7月15日）。

③ TLS Client Puzzles Extension，https://datatracker.ietf.org/doc/draft-nygren-tls-client-puzzles/（E. Nygren，2015年7月）。

④ SSL computational DoS mitigation，http://vincent.bernat.im/en/blog/2011-ssl-dos-mitigation.html（Vincent Bernat，2011年11月1日）。

你拥有的资源越多，攻击者对你进行攻击的成本就越高。资源超配是昂贵的，但如果你经常受到攻击，购买更多的服务器并且拥有非常大的网络连接是一个可行的方法。

- 第三方援助

  当其他一切都失败时，你可以通过聘请专门从事缓解分布式拒绝服务攻击的公司来处理问题。他们的主要优势是拥有充足的资源以及技术诀窍。

所有这一切并不意味着你应该放弃调整TLS以尽可能减少被拒绝服务攻击。与此相反，TLS的某些方面使拒绝服务攻击更加容易；它们需要你的关注。

9.3.1　密钥交换和加密 CPU 开销

使用明文协议（例如HTTP）的时候，服务器通常在将文件发送给客户端上耗时最多。这是很常见的操作，应用可以让内核向套接字发送特定的文件而不需要关心具体细节。使用TLS，相同的应用需要读取文件、加密然后发送。这样总会变慢。

客户端同样也会变慢，因为需要以相反的顺序执行相同的操作。最为复杂的是握手，需要多次CPU密集型的加密操作。客户端和服务器由于握手期间的不同密钥交换算法，会有不同的性能概况，也有不同的时间消耗。如果客户端需要执行比服务器更少的工作，那就有可能导致拒绝服务攻击。

使用RSA正是这样的例子，典型使用方式（使用公共指数）通过公钥（客户端执行）操作比通过私钥操作（服务器）要快。实际上，平均2048位的RSA密钥，服务器最终需要做4倍的工作。结果就是一个CPU中等的客户端可以通过并发执行多个握手压跨硬件配置强劲的服务器。

为了确认这一点，我通过两台相同配置的电脑做了测试，一台运行2048位RSA私钥的Web服务器，另一台去攻击它。我用流行的压测工具ab很轻松地将服务器CPU压垮。在此期间，客户端运行的CPU消耗只是略微超过10%。

RSA依然是最主要的身份验证和密钥交换算法，但有一个好消息：它在被逐步淘汰。最大的问题是它不支持前向保密。在短期内，人们正转向ECDHE_RSA，保留RSA用于身份验证但使用ECDHE用于密钥交换。使用ECDHE_RSA，客户端依旧执行更少的工作，只是没那么糟糕，只有不到2.5倍。此外未来将会使用ECDHE_ECDSA，之前的情况会出现反转，客户端多执行1.5倍工作。

> **注意**
>
> 要从这些算法变化里受益，你需要将RSA密钥交换从你的配置中删除。否则攻击者在攻击时可以强制使用最慢的密码套件。

加密也是有开销的。在本章前面部分你知道SEED加密比最常使用的AES-128加密要慢4倍，3DES比AES-128要慢11倍。许多服务器为了例如IE6这样的旧版本的客户端才保留3DES配置。尽管密码套件的选择在TLS拒绝服务攻击中不可能起主要作用，但它绝对会让事情变得更糟。

9.3.2　客户端发起的重新协商

重新协商（renegotiation）是一个协议功能，允许一方请求一次新的握手以协商不同的连接参数。这个功能很少被用到，允许客户端请求重新协商，目前虽然没有任何实际的用途，但使得缓解拒绝服务更加困难。

在“标准的”TLS计算拒绝服务攻击中，每个连接只有一次握手。如果在你掌控的地方有连接限制，你知道每个到TLS服务器的连接会消耗一定的CPU处理能力。如果允许客户端发起的重新协商，攻击者可以在同一个连接上执行多次握手，绕过检测机制。[①]这个技术同时减少了需要的并发连接的数量，从而降低了整体攻击的延迟。

2011年10月一个名为The Hacker’s Choice的德国黑客组织发布了一个叫作`thc-ssl-dos`的工具，使用重新协商放大对TLS的拒绝服务攻击。[②]

并非所有服务器都支持客户端发起的重新协商。IIS在IIS6之后停止支持，Nginx从未支持过，Apache在2.2.15停止支持。但仍有许多厂商不愿意删除此功能。保留客户端发起的重新协商功能的一些厂商，正在寻求方法限制同一连接重新协商的数量。理想情况下，你不应该允许客户端发起重新协商。

9.3.3　优化过的 TLS 拒绝服务攻击

重新协商让TLS拒绝服务攻击更难被检测到，但是使用它的工具并没有本质的区别：它们都是向网站潜在发送大量虚假的客户端连接。关于这两种场景，握手CPU消耗不对称是让攻击成为可能的原因。事实证明，通过优化可以让客户端不需要进行加密操作。

`thc-ssldos`工具一经公布就受到了广泛的媒体关注。Eric Rescorla是TLS协议的设计者之一，跟进分析使用重新协商放大拒绝服务的技术。[③]他的结论是还有一个更简单的方式执行TLS拒绝服务攻击。他的办法是客户端使用无需加密操作的硬编码的握手消息。此外，避免解析或者验证任何从服务器收到的消息。因为消息结构上是正确的，但直到握手的最后服务器才验证出来，这时已经太晚了，所有耗时的工作都已经完成。

使用Eric的蓝图，Michal Trojnara随后写了一个叫作`sslsqueeze`的概念验证工具。[④]

当我测试`sslsqueeze`的时候，我发现它比`ab`的表现要好很多。我在一台2.80 GHz Inte Xeon E5-2680单核的机器上安装它，目标是同一个数据中心8核CPU的服务器。这个工具操作几秒后就把目标服务器上全部的CPU资源都耗尽了。

---

① 仍然可以检测到攻击，但通常需要深层流量监听，最好通过解析协议消息。这种能力就没有连接计数那样普遍了。

② THC SSL DoS，https://thehackerschoice.wordpress.com/2011/10/24/thc-ssl-dos/（The Hacker’s Choice，2011年10月24日）。

③ SSL/TLS and Computational DoS，http://www.educatedguesswork.org/2011/10/ssltls_and_computational_dos.html（Eric Rescorla，2011年10月25日）。

④ ftp://ftp.stunnel.org/sslsqueeze/的索引（Michal Trojnara，2011年11月16日）。

第 10 章

HTTP严格传输安全、内容安全策略和钉扎

本章论述几个可以大幅提升SSL/TLS和PKI生态系统安全性的技术，我们将其分成两组。第一组的HTTP严格传输安全（HTTP strict transport security，HSTS）和内容安全策略（content security policy，CSP）都是HTTP专有的，并且得到了浏览器的广泛支持。它们不仅在当前很实用，而且也是网站安全的基础。我会尽可能详尽地介绍它们的部署。

第二组技术是为了TLS服务器身份验证更加安全而实现的钉扎（pinning）。有几个成熟度不同的方法竞争实现钉扎。对于客户端和服务器通信都由相同开发人员控制的本机应用程序，钉扎已经得到充分实践。对于浏览器，一些钉扎的办法也可以马上部署，但需要手动操作，并与浏览器厂商协调。存在一些新的标准，以及未来可能证明是可行的有潜力的新方向。本章涵盖了所有可用的选项。

10.1 HTTP 严格传输安全

2012年11月发布为RFC 6797[①]的HTTP严格传输安全（HTTP strict transport security，HSTS），是一项描述严格方法处理网站加密的提议标准。它设计用于缓解当前浏览器中关于TLS实现的几个关键弱点。

- 无法知道网站是否支持TLS

  HTTP没有指定能让浏览器判断网站是否实现了TLS的方式。[②]因此，在地址栏中输入一个没有指定协议的URL时，浏览器必须在HTTP和HTTPS协议之间作选择。当前，浏览器默认使用容易被截获的明文通信。
- 证书容错问题

  在Web刚刚起步时，浏览器已经回避了TLS连接真实性的问题。浏览器不是放弃与无效证书站点的连接，而是显示警告信息，并允许用户点击跳过，继续访问。有研究表明，许

① RFC 6797: HTTP Strict Transport Security (HSTS)，http://tools.ietf.org/html/rfc6797（Hodges和Jackson，2012年11月）。

② 这可以使用DNS SRV记录实现，这些记录设计为指向提供某个特定服务的确切主机名和端口。SRV记录是在2000年2月发布的RFC2782中定义的。

多用户忽略了警告，将自己暴露在主动攻击之下。

- 混合内容问题

  开发安全网站时有一个常见错误是，在一个安全的HTML页面中引入了明文资源。所有浏览器在某种程度上都允许此类资源，并且在很多情况下这些明文连接可能用于危害整个用户会话。另一个常见问题是，将同一个域名上的明文页面和加密页面混合到一起。这很难正确实现，往往会导致网站的漏洞。

- Cookie安全问题

  另一个常见的实现错误是，忘记保护应用程序Cookie的安全。即使是只允许TLS访问的网站，主动网络攻击者也可以从受害者的浏览器窃取Cookie。

> **注意**
> 有关上述问题的完整讨论和不同的攻击方式，请阅读第5章。

在将HSTS部署到网站时，它通过两个机制解决以上全部问题：(1) 明文URL被透明重写成使用加密；(2) 全部的证书错误被视为致命的（不允许用户点击跳过）。通过这种方式，HSTS极大减少了攻击面，从而让部署安全网站变得更加容易。这是TLS近期最棒的一件事情了。

HSTS的产生源于Jackson和Barth在2008年的成果，他们设计了一种基于Cookie的强制HTTPS机制[1]，允许高级用户对一些支持HTTPS的不安全网站透明地增强安全性。在他们的论文中，以Firefox插件的方式提供概念验证。

10.1.1　配置 HSTS

通过在加密的HTTP响应中包含`Strict-Transport-Security`头实现网站HSTS，像下面这样配置：

```
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
```

假设TLS连接没有错误，兼容的浏览器将会在`max-age`参数指定的保留期内激活HSTS。`includeSubDomains`参数指定应该在发出头的主机及其所有子域上启用HSTS。

> **警告**
> 在部署HSTS启用`includeSubDomains`参数之前，确认如果强制浏览器对整个域名空间使用加密，可能对共享同一域名的其他网站造成负面影响。至少要确保所有网站真正支持加密，并拥有有效的证书。

规范要求用户代理对明文连接或者有证书错误（包括自签名证书）的加密连接忽略HSTS响应头。这主要是为了防止主动网络攻击者轻易对只支持明文访问的网站发起拒绝服务攻击。此外，不允许对IP地址使用HSTS。

通过`Strict-Transport-Security: max-age=0`这样将`max-age`参数设置为零来撤销HSTS是可行

[1] ForceHTTPS: Protecting High-Security Web Sites from Network Attacks，https://crypto.stanford.edu/forcehttps/force-https.pdf（Jackson和Barth，2008）。

的，但是只有当浏览器（对之前启用HSTS的网站）再次访问网站并且更新配置时撤销才会生效。因此撤销是否成功（HSTS策略调整）取决于用户的访问频率。

最佳情况下，应该将HSTS配置在离用户最近的位置。例如，你有很多Web服务器和前端反向代理（应用程序或者防火墙），最好只在前端配置HSTS。否则就在Web服务器层配置HSTS。如果Web服务器不明确支持HSTS，很可能有允许任意增加响应头的机制。后者也可以很好地运作，但要阅读附属细则。有些情况下对出错响应（例如404页面）添加响应头是不行的，或许需要特殊的配置。

如果其他方法都失败了，可以在应用程序层面增加HSTS。但需要注意你的应用程序并不是对网站的所有请求都是可见的。例如，Web服务器通常直接提供静态资源，并且自己处理一些重定向。

10.1.2 确保主机名覆盖

默认情况下，仅会在发出`Strict-Transport-Security`响应头的主机名上启用HSTS。在不只一个主机名（例如，store.example.com和accounts.example.com）上部署的网站应该注意对全部主机名都启用HSTS。否则可能出现用户访问一些并没有配置HSTS的主机名而得不到保护。

有些应用程序使用所谓的域Cookie，一种在根域（例如example.com）上设置并且可以被任一子域使用的Cookie。这种技术通常用于跨多个主机名并且需要统一验证和会话管理的网站。这种场景下对全部主机名包括根域名启用HSTS尤为重要。你不会希望遗留任何可能被攻破的漏洞。

即使只使用一个主机名的网站也需要考虑这个问题，因为可能用户有时访问网站不带前缀（例如example.com），有时又带前缀（例如www.example.com）。由于我们不控制入站链接，必须在配置HSTS时特别小心，以确保对全部主机名都启用。

10

> **警告**
> 一个常见错误是忘记对重定向配置HSTS。例如，一些用户可能首先访问你的根域名（例如example.com）。如果你没有对重定向配置HSTS，尽管对主域名配置了HSTS,，用这种方式访问的用户还是可能遭受SSL剥离攻击。为了达到最佳效果，列举能访问到你的网站的全部路径，并且全部加上HSTS。

10.1.3 Cookie 安全

因为HSTS强制对特定网站的全部连接进行加密，你可能认为即使不安全的Cookie也能安全对抗主动网络攻击者。不幸的是，Cookie规范是比较随意的，这为额外的攻击向量创造了机会，比如以下几种情况。

- 通过伪造的主机名攻击

  Cookie通常是为某个个特定的主机名及其所有子域设置的。同时，主动网络攻击者可以在需要的时候窜改DNS，创建目标网站同一域名下的任意主机名。因此如果你对www.example.com设置Cookie，攻击者可以通过强制拦截访问madeup.www.example.com来

窃取Cookie。如果Cookie是不安全的，明文访问就可以实现这种攻击了。如果Cookie是安全的，攻击者可以配置自签名证书，祈祷用户会忽略浏览器警告，点击通过。

- Cookie注入

  Cookie规范未对安全的Cookie使用独立的命名空间。这意味着一个来自明文连接的Cookie设置可以覆盖一个已经存在的安全Cookie。在实践中，这意味着一个主动网络攻击者可以对其他安全的应用程序注入任意Cookie。

  在域Cookie的场景中，攻击者可以从现有同级主机名（例如blog.example.com）注入Cookie。否则主动网络攻击者可以伪造一个任意主机名并且从它注入。

通过使用`includeSubDomains`参数在交付主机名及其所有子域时激活HSTS，可以在很大程度上解决上述问题。当使用域Cookie的时候，唯一安全的途径是对根域名激活HSTS以便覆盖整个域名空间。我在5.3节中详细论述了Cookie安全问题。

10.1.4　攻击向量

HSTS极大提升了我们保护网站安全的能力，但仍有一些边缘情况值得你留意。考虑以下几种情形。

- 首次访问

  因为HSTS是通过设置HTTP响应头激活的，对首次访问不提供任何安全保障。可是一旦HSTS被激活就可以持续保护直到过期。首次访问缺乏安全的问题可以通过在浏览器内嵌（或预加载，preloading）一份已知支持HSTS的网站清单来缓解。这种方法可行的原因是支持HSTS的网站数量仍然很小。

- 较短保存时间

  HSTS最好部署较长的保存时间（例如，最少6个月）。如此一来，不仅在用户在首次会话期间得到保护，而且用户对网站的后续访问都持续受到保护。如果保存时间很短，用户在过期之前没有再次访问，那他们下次访问时就得不到保护。

- 时钟攻击

  电脑被配置使用无验证的网络时间协议（network time protocol，NTP）自动更新时钟的用户，可能被主动网络攻击者覆盖NTP消息进行攻击。通过将电脑时钟设置成未来某一时刻可以导致网站的HSTS策略失效，使受害者下次访问变得不安全。这种攻击向量的危险性取决于NTP访问频率。这通常是每天一两次。按照2014年10月发布的调查报告，操作系统受这种攻击的难易程度相差很大。有些系统，比如Fedora每分钟都要同步时钟，导致很容易成为攻击的目标。其他比如OS X同步频率要低一些（分钟级别），但仍然相对容易攻击。似乎Windows是最安全的，尽管也使用未验证的NTP，但只有每周同步一次，并且内置防护措施防止大范围的时间调整。①

① Bypassing HTTP Strict Transport Security，https://www.blackhat.com/docs/eu-14/materials/eu-14-Selvi-Bypassing-HTTP-Strict-Transport-Security-wp.pdf（Jose Selvi，2014年10月）。

- 响应头注入

  响应头注入是一种让攻击者对受害者流量注入任意响应头的Web应用程序漏洞。一旦应用出现这种漏洞，攻击者可以注入一个伪造的Strict-Transport-Security响应头从而禁用HSTS。对未使用HSTS的应用程序，这种攻击可用于启用HSTS并可能造成DoS攻击。

  当攻击者对已经启用HSTS的应用程序实施这种攻击的时候，出站响应头会包含两份Strict-Transport-Security字段。如果攻击者设置的字段排在响应头的前面，浏览器就会优先使用它。

- TLS截断

  尽管TLS协议不会被截断攻击利用，但多数浏览器的实现却可能导致这种风险。技术精湛的主动网络攻击者可以使用特殊的技术拦截TLS连接，并且在max-age参数的第1位截断。这种攻击一旦成功，就可以减小HSTS的持续时间，最多只能是9秒。这就是我在6.7.2节中描述的所谓Cookie截断攻击。

- 混合内容问题

  HSTS的设计者选择了不完全解决混合内容问题，可能因为这是一个难题，并且浏览器厂商倾向于使用不同的方法处理它。结果就是HSTS包含了对12.4部分（不允许加载混合内容）唯一的非规范意见。

  HSTS仍旧提供了部分解决方案，来自同一主机名（启用HSTS的地方）的明文请求是不允许的。为了解决第三方混合内容问题，要部署内容安全策略（content security policy，CSP）。CSP可以用于只允许来自特定页面的HTTPS请求。

- 主机名和端口共享

  HSTS是对整个主机名和全部端口生效的。这种方式在多方可以控制响应头的共享托管情况下不能很好工作。在这种情况下，应小心地筛选所有响应，以确保发送正确的HSTS头（或者根本不发送HSTS头）。

10.1.5　浏览器支持

HSTS目前能在桌面浏览器得到正式的支持，要感谢早在2010年和2011年分别被Chrome和Firefox所接受。在2013年后期Safari浏览器的OS X10.9版本也加入了其中。Internet Explorer 11于2015年6月为其Windows 7和8.1平台添加了对HSTS的支持（参见表10-1）。

表10-1　浏览器对HTTP严格传输安全的支持

浏　览　器	HSTS支持	开始支持的版本	预　加　载
Chrome	支持	v4.0.249.78；2010年1月[a]	支持
Firefox	支持	v4；2011年3月[b]	支持（v17+）
Internet Explorer	支持	v11（Windows 7和 8.1）；2015年6月[c]	支持
Opera	支持	v12（Presto/2.10.239）；2012年6月[d]	支持（v15+）
Safari	支持	v7（OS X 10.9 Mavericks）； 2013年10月	支持

a Stable Channel Update，http://googlechromereleases.blogspot.com/2010/01/stable-channel-update_25.html（Chrome版本博客，2010年1月25日）。

b Firefox 4 release notes，http://website-archive.mozilla.org/www.mozilla.org/firefox_releasenotes/en-US/firefox/4.0/releasenotes/（Mozilla，2011年3月22日）。

c HTTP Strict Transport Security comes to Internet Explorer11 on Windows 8.1 and Windows 7，https://blogs.windows.com/msedgedev/2015/06/09/http-strict-transport-security-comes-to-internet-explorer-11-on-windows-8-1-and-windows-7/（Microsoft Edge Dev Blog，2015年6月9日）。

d Web specifications support in Opera Presto 2.10，http://www.opera.com/docs/specs/presto2.10/#m210-244（Opera，检索于2014年4月19日）。

大多数浏览器出厂就预加载了已知支持HSTS的站点列表，这使得它们在第一次连接到站点时是完全安全的。最初，这些列表都是手动编译或者通过远程扫描的。之后，Google开始运行一种中心预加载注册服务，现在所有浏览器供应商都依赖这些服务。[①]但是，关于各个供应商如何更新其预加载列表的细节却很少。

10.1.6　强大的部署清单

尽管HSTS比较简单，如果你工作的环境比较复杂的话，部署它可能会相当复杂。针对除了最简单环境之外的情况，我建议将部署HSTS分为两个主要步骤：首先，开始一项测试运行，该运行按配置让一切顺利进行，除了使用非常短的持续时间值。之后，增加持续时间到所期望的值。

按照以下步骤进行验证。

(1) 确保`Strict-Transport-Security`头是在全部主机名（例如，accounts.example.com和www.example.com）的所有加密响应上发出的，同时指定`includeSubDomains`参数。

(2) 确保在根域名（例如，example.com）上启用HSTS，同时指定`includeSubDomains`参数。

(3) 确定指向你的网站的全部路径，并仔细检查确保所有重定向都发出HSTS策略。

(4) 一开始先用一个短暂的过期时间作为临时策略。万一忘记在生产环境有一个只支持明文访问的重要网站，这会让你相对容易地恢复回去。

(5) 重定向所有HTTP通信到HTTPS。这将确保用户始终在第一次访问时收到HSTS指令。

(6) 修改你的网站使每个主机名至少向根域名提交一个请求。这将确保HSTS是完全对整个域命名空间启用，即使用户不直接访问根域名。

(7) 如果在你的网站面前有一个反向代理，在代理级别集中配置HSTS策略是一个加分项。为了防止利用头注入漏洞绕过HSTS，请删除由后端Web服务器设置的任何HSTS响应头。

经过一段时间，在确认各方面部署都正确时，增加策略的保留时间。可以逐步增加或者可以立即切换到长期值。具体步骤如下所示。

(1) 将策略保持持续时间增加为一个长期的值，例如，12个月。这不仅会给你最好的保护，而且能够确保你满足预加载列表的最低期限要求。

(2) 遵守预加载指令的要求，并通知网站hstspreload.appspot.com的预加载列表维护人员。

① Google's HSTS preload registration service，https://hstspreload.appspot.com/（Adam Langley，检索于2015年7月26日）。

> **如果你无法对整个域名激活，HSTS将会怎样？**
>
> 为了达到最佳效果，应该对主域名及其所有子域名启用HSTS。不幸的是，这并不总是可能的。特别是如果你使用了大量现有的基础设施，可能需要一段时间才能把所有的服务迁移到HTTPS。
>
> 即使在这种情况下，你仍然只对主应用程序主机名（例如www.example.com，而不是example.com）使用`includeSubDomains`。它将提供足够的安全性，除了使用域Cookie的情况。但是，需要仔细做到这一点。因为HSTS策略里并不包括其适用的主机名名称，有可能会无意中在错误的地方激活HSTS。
>
> 当部署不覆盖任何子域的HSTS时，会存在10.1.3节中描述的风险。这种风险可以通过部署加密安全机制进行缓解，从而保证Cookie的机密性和完整性。

10.1.7 隐私问题

HSTS的性质决定了浏览器使用持久性存储保持跟踪用户访问的HSTS网站。当用户首次遇到某个HSTS站点时，会将一个项添加到浏览器的HSTS数据库中。这一事实使得它可以测试某人之前是否访问过特定网站，只要让他们点击一个网站的明文链接即可。如果他们访问的就是明文链接，说明他们以前从未访问过该网站。但是如果他们之前已经访问过该网站，HSTS会生效，重定向链接，访问会变成HTTPS（取代HTTP）。

在本质上一个HSTS策略可以用于存储浏览器信息中的1位。1位看起来并不多，但是使用泛域名证书时，对手会制造尽可能多的主机名，每个主机名需要一个单独的HSTS策略，每个都携带一位的信息。[①]

10.2 内容安全策略

内容安全策略（content security policy，CSP）是一种声明的安全机制，可以让网站运营者能够控制遵循CSP的用户代理（通常是浏览器）的行为。通过控制要启用哪些功能，以及从哪里下载内容，可以减少网站的攻击面。

CSP的主要目的是防御跨站点脚本（cross-ste scripting，XSS）攻击。例如，CSP可以完全禁止内联的JavaScript，并且控制外部代码从哪里加载。它也可以禁止动态代码执行。禁用了所有的这些攻击源，XSS攻击变得更加困难。

CSP由Mozilla开发，曾经有几年试验过概念，一开始称为内容限制（content restriction）[②]，

① The Double-Edged Sword of HSTS Persistence and Privacy，http://www.leviathansecurity.com/blog/the-double-edged-sword-of-hsts-persistence-and-privacy/（Leviathan Security Group，2012年4月4日）。

② Content Restrictions，http://www.gerv.net/security/content-restrictions/（Gervase Markham，最后更新于2007年3月20日）。

后来改称为内容安全策略。[①]2012年11月CSP 1.0成为W3C的候选议案[②]；当前CSP 1.1相关工作还在开展中。

一个网站通过设置Content-Security-Policy响应头启用所需的CSP策略。[③]为了便于读者理解，下面给出一个策略示例：

```
Content-Security-Policy: default-src 'self'; img-src *;
                         object-src *.cdn.example.com;
                         script-src scripts.example.com
```

该策略默认只允许资源从同一来源加载，但允许图像从任何URI加载，插件内容只从指定的CDN地址加载，外部脚本仅从scripts.example.com加载。

不像HSTS，CSP策略不是持久的；只在引用它们的页面上生效，之后就失效了。因此，使用CSP风险要小很多。如果产生错误，策略可以立即更新生效。即使被注入响应头，也不会有持续性拒绝服务攻击的风险。

10.2.1　防止混合内容问题

如果安全网页依赖的资源（例如，图像和脚本）是通过明文连接检索的，就会出现混合内容问题。这些年浏览器虽然对这个问题的处理有改善，但它们的方法通常仍然太过宽松。例如，所有浏览器都允许所谓的*被动式混合内容*（passive mixed content），通常是图像。不出所料的是，在不同浏览器上也有处理差异。例如，Safari目前不强加任何限制，甚至对脚本也是。有关混合内容问题的详细讨论，可以查阅5.8节。

因为CSP让我们能够控制其中的内容来源，所以可以用它来指导遵循CSP的浏览器只使用安全协议，也就是对WebSocket协议用wss，剩下的其他都用https。

因此，只解决混合内容问题，而不尝试改善其他的话，可以考虑以下面的CSP策略作为出发点：

```
Content-Security-Policy: default-src https: 'unsafe-inline' 'unsafe-eval';
                         connect-src https: wss:
```

该策略包括以下三个主要内容。

- default-src指令确定该页面可以从任何地方（任何主机和任何端口）加载提供了安全连接（https）的内容。
- 'unsafe-inline'和'unsafe-eval'表达式重新启用内联JavaScript和动态代码执行，这些默认情况下都是被CSP禁用的。理想情况下，你不会希望在策略里保留这些表达式，但没有它们大多数现有的应用都会被阻断。
- connect-src指令控制脚本接口（比如XMLHttpRequest、WebSockets、EventSource等）所

---

① Content Security Policy，http://people.mozilla.org/%7Ebsterne/content-security-policy/（Mozilla’s CSP Archive，最后更新于2011年）。

② Content Security Policy，http://www.w3.org/TR/CSP/（W3C Candidate Recommendation，检索于2015年7月）。

③ 你可能会看到在博客文章中提到的其他标题名称，例如，X-Content-Security-Policy和X-Webkit-CSP。这些头存在于CSP的初期，当时的功能主要是实验使用。今天唯一应用的头名称是官方的这个。

使用的内容位置。

一旦确立了这个最初的策略可以工作，请考虑收紧JavaScript执行（通过删除'unsafe-inline'和'unsafe-eval'表达式），并用更具体的主机取代通用源的限制（例如，使用https://cdn.example.com取代https:）。

10.2.2 策略测试

有关CSP的一个好处是，能够在执行一个策略的同时并行测试其他策略。这意味着，你甚至可以在比开发环境要复杂得多的生产环境中部署测试策略。

Content-Security-Policy-Report-Only响应头用于创建一个仅用于测试的策略：

```
Content-Security-Policy-Report-Only: default-src 'self'
```

如果只启用报告的策略出错，并不会阻塞请求，但可以配置报告使得故障可被传递回起初的网站。

10.2.3 报告

CSP的另一个特点是支持报告，可以用于跟踪违反策略的行为。有了这个功能，部署就变得容易多了。这也使得很容易知道部署在生产策略中是否阻断其他资源。

要想启用报告，请使用report-uri指令：

```
Content-Security-Policy: default-src 'self';
                         report-uri http://example.org/csp-report.cgi
```

这样违反CSP策略的行为将被提交给指定的URI，使用POST方法请求并将数据附在请求体上报。例如：

```
{
  "csp-report": {
    "document-uri": "http://example.org/page.html",
    "referrer": "http://evil.example.com/haxor.html",
    "blocked-uri": "http://evil.example.com/image.png",
    "violated-directive": "default-src 'self'",
    "original-policy": "default-src 'self'; report-uri http://example.org/csp-report.cgi"
  }
}
```

10.2.4 浏览器支持

当前浏览器对CSP的支持较好。Chrome和Firefox试验了多年，而且最近其他主流浏览器也逐步开始支持（参见表10-2）。桌面浏览器里面唯一不支持CSP的是Internet Explorer。Microsoft的下一代浏览器称为Edge，在Windows 10中首次亮相，支持CSP 1.0。①

① 兼容性，https://msdn.microsoft.com/en-us/library/dn904497%28v=vs.85%29.aspx#content_security_policy_1.0（*Microsoft Edge Developer Guide*，检索于2015年7月）。

表10-2　浏览器对内容安全策略的支持

浏览器	CSP支持	开始支持的版本
Android浏览器	支持	4.4.x（2013年10月）[a]
Chrome	支持	v25（2013年2月）。[b] 从2011年6月开始提供实验性支持[c]
Firefox	支持	v23（2013年8月）。[d] 从2009年6月开始在Firefox v4中提供实验性支持[e]
Internet Explorer	不支持[f]	–
Opera	支持	v15（2013年7月）
Safari	支持	v7（在2013年9月在iOS 7中，2013年10月在OS X 10.9中）。从v6开始在Mountain Lion中提供实验性支持[g]

a Content Security Policy，http://caniuse.com/contentsecuritypolicy（Can I use，检索于2014年6月29日）。

b Chrome 25 Beta: Content Security Policy and Shadow DOM，http://blog.chromium.org/2013/01/content-security-policy-and-shadow-dom.html（The Chromium Blog，2013年1月14日）。

c New Chromium security features, June 2011，http://blog.chromium.org/2011/06/new-chromium-security-features-june.html（The Chromium Blog，2011年6月14日）。

d Content Security Policy 1.0 lands in Firefox Aurora，https://hacks.mozilla.org/2013/05/content-security-policy-1-0-lands-in-firefox-aurora/（Mozilla Hacks，2013年5月29日）。

e Shutting Down XSS with Content Security Policy，https://blog.mozilla.org/security/2009/06/19/shutting-down-xss-with-content-security-policy/（Brandon Sterne，Mozilla Security Blog，2009年6月19日）。

f Windows 10的新版浏览器Microsoft Edge提供了对CSP 1.0的支持。

g Safari 6 gets Content-Security-Policy right，http://rachelbythebay.com/w/2012/07/29/csp/（rachelbythebay，2012年7月29日）。

10.3　钉扎

钉扎（pinning）是一种安全技术，用于将服务与一个或者多个密码身份（例如证书和公钥）进行关联。根据钉扎在哪里使用以及如何使用，可用于实现以下三个主要安全改进。

- 减小攻击面

  目前主要被使用的TLS身份验证模式依赖公共CA。他们的工作是给域名的持有者颁发证书，而不是给其他人。反过来，用户代理无条件信任所有CA颁发的证书。这种模式具有一个巨大的缺陷：颁发证书不需要站点所有者的授权。其结果是，任何CA可以颁发任何域名的证书。由于有数百个CA甚至数千的实体会以这样或者那样的方式影响证书颁发，攻击面是巨大的。

  使用钉扎，持有者可以指定（PIN）允许哪些CA颁发证书给自己的域名。他们可以观察市场，决定哪一个或两个CA最适合自己，并相应地配置钉扎。之后他们不再关心数以百计的公共CA，因为不再存在风险。

- 密钥连续性

  密钥连续性（key continuity）是前面一种使用场景的变化，但它可以不依赖于公共CA而使用。我们假设你以某种方式知道一些网站的某个特定密钥是有效的。如此一来，每次你访问网站时，可以把它们当前的密钥与你的“正确”密钥进行比较；如果密钥匹配，你知道自己没有受到攻击。

密钥连续性最常见的是用于SSH协议。当第一次访问时将密钥与服务器相关联，并在以后的访问中检查。这也被称为*首次使用信任*。

Firefox在证书无法验证是否允许你创建一个例外的时候使用密钥连续性,并且仅对特定证书。如果你后续受到不同（MITM）证书的攻击，Firefox会显示一个证书并且再次警告。

- *身份验证*

  钉扎甚至可以用于身份验证，基于可靠的（安全）信道进行通信时提供所需要的加密标识给最终用户。例如，如果我们部署了一个无法通过主动网络攻击破坏的安全DNS，那么就可以用它来存储网站证书的指纹。这些指纹可以在访问每个网站时作检查。

10.3.1 钉扎的对象

钉扎可以用于多个加密元素，通常的选择是证书和公钥。例如，一个可能的方法是你保存期望访问的特定网站的一个或者多个证书的列表，每当你访问网站时就可以将实际访问得到的证书与保存的证书列表相比较。通过这种方法，很容易就检测出伪造的证书。由于证书不会改变，通过它们的散列值（例如，SHA256）进行跟踪会更容易实现。

在实践中，公钥钉扎更加实用，因为证书有时重新签发并不会改变公钥。多个证书使用相同的公钥也是常见的。因此，如果你对公钥钉扎，可以让与它相关联的所有证书都正常运作。

虽然协议不直接依赖证书公钥钉扎，但对于TLS来说钉扎最好的元素是X.509证书的SubjectPublicKeyInfo（SPKI）字段。[①]这个字段包含公钥本身以及进行准确识别所必需的额外元数据：

```
SubjectPublicKeyInfo ::=   SEQUENCE {
     algorithm            AlgorithmIdentifier,
     subjectPublicKey     BIT STRING }
```

如果想检查指定证书的SubjectPublicKeyInfo字段内容，可以使用以下命令：

```
$ openssl x509 -in server.crt -noout -text
[...]
Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:b8:0e:05:25:f8:81:e9:e7:ba:21:40:5f:d7:d4:
                    09:5c:8c:d4:e9:44:e7:c0:04:5b:7f:6e:16:8a:01:
                    37:2b:b9:ed:b6:09:cd:1f:55:d5:b8:ee:79:13:ae:
                    e7:1d:6a:ec:01:7c:02:5a:10:af:f9:68:28:ff:d5:
                    61:b0:37:f8:a6:b2:87:42:90:3c:70:19:40:67:49:
                    99:1d:3c:44:3e:16:4e:9a:06:e4:06:66:36:2f:23:
                    39:16:91:cf:92:56:57:1d:30:db:71:5a:68:a2:c3:
                    d5:07:23:e4:90:8e:9e:fb:97:ad:89:d5:31:3f:c6:
                    32:d0:04:17:5c:80:9b:0c:6d:9b:2a:b2:f9:39:ac:
                    85:75:84:82:64:23:9a:7d:c4:96:57:1e:7b:bf:27:
```

① 有关X.509证书结构的更多信息，请参考3.3节。

```
                2e:48:2d:9e:74:90:32:c1:d8:91:54:12:af:5a:bb:
                01:20:15:0e:ff:7b:57:83:9d:c2:fe:59:ce:ea:22:
                6b:77:75:27:01:25:17:e1:41:31:4c:7f:a8:eb:0e:
                8c:b9:18:b2:9a:cc:74:5e:36:1f:8f:a1:f4:71:a9:
                ff:72:e6:a0:91:f0:90:b2:5a:06:57:79:b6:1e:97:
                98:6b:5c:3a:a9:6a:be:84:bc:86:75:cb:81:6d:28:
                68:c0:e5:d5:3e:c5:f0:7d:85:27:ae:ce:7a:b7:41:
                ce:f9
            Exponent: 65537 (0x10001)
```

要生成一个散列SPKI，首先将字段从证书提取到新文件：

```
$ openssl x509 -in server.crt -noout -pubkey | \
  openssl asn1parse -inform PEM -noout -out server.spki
```

接入你可以计算它的SHA256散列值，并使用Base64进行编码：

```
$ openssl dgst -sha256 -binary server.spki | base64
zB8EXAKscl3P+4a5lFszGaEniLrNswOQ1ZGwD+TzADg=
```

10.3.2　在哪里钉扎

谈到决定在哪里钉扎，答案并不明确。最明显的选择是对服务器证书的公钥钉扎，但这种方法有几个缺点。其中之一是服务器天然就容易受到攻击。如果服务器的私钥被泄露，此时用新的私钥进行替换，那么旧的钉扎将不再有效。即使没有攻击，服务器密钥也应经常轮转，以最小化使用相同密钥保护的数据量。最后，通常复杂部署的同一个站点依赖于多个密钥和证书，对所有这些进行钉扎将会很困难并且费时。

出于这个原因，我们可以考虑钉扎在证书链的其他地方。当前大多数证书链以最终实体证书开始，之后有一个中间CA证书，最后是根证书。如果对后两者中的一个作钉扎，应该能够改变服务器的身份，从相同CA得到新的证书，同时继续使用相同的钉扎。

这听起来很理想，但也有一些问题。首先，CA通常有多个根。他们也有多个中间CA用于不同类型的证书，为了将风险降至最低，会更改签名算法，等等。下一次你从同一CA得到的证书有可能会使用不同的中间证书和根证书。

此外，为了支持旧版客户端，CA还依赖其他更为成熟的CA的根进行交叉认证。这意味着，一个给定的证书可能有多个有效的信任路径。在实际应用中，用户代理可以决定使用与你所想路径不同的信任路径。如果发生这种情况，并且你的钉扎关联到一个被排除信任的路径，验证将失败。因此钉扎的最佳选择是第一个中间CA证书。由于它的签名是对最终实体证书，颁发CA的公钥必须始终在证书链中。这种方法确保用户浏览器将无法绕过钉扎，但未来CA仍然有可能使用不同的中间CA签发证书。虽然没有明确的解决办法，但可以采取下面这两个降低风险的措施。

- 要求你的CA支持钉扎，并承诺确保你的钉扎在未来的证书中保持有效。
- 始终有来自不同CA的可选证书和相应钉扎。

使用钉扎最高效的运营方式是用自己的中间CA。通过这个设置，相同的钉扎对你全部的证书都有效，并且你知道新的证书会用相同的父证书签发。不幸的是，这种方式虽然高效但也非常

昂贵，很少有人能承担获取并管理自己的中间CA的费用。

最安全的方式是只对你必须访问的密钥进行钉扎。这能确保其他人无法生成通过钉扎验证的有效证书。对大多数网站而言，这意味着直接对服务器证书钉扎（并接受因此带来的所有其他困难）。

不论你决定使用哪种方式，至少保留一个钉扎有效的备份证书链。这是从密钥丢失或者钉扎失败中优雅恢复的唯一途径。

10.3.3　应该使用钉扎吗

钉扎是一种减小攻击面的强大技术，但也是有代价的。要部署钉扎，你需要很好地掌握权衡，并且需要一个成熟的组织来应对运营它的挑战。很明显的问题是钉扎仅确保对所钉扎的身份建立TLS连接。不管出于什么原因，如果你失去了这些身份，会发生什么？

担心自己造成的拒绝服务攻击可能是钉扎一直发展缓慢的原因。浏览器厂商明白这一点，从钉扎提案中也可以显而易见地看出来。不同于HSTS常见的长期策略保留期限（比如一年），钉扎周期通常以天计。

然而，如果你认为自己的网站可能会通过某个以欺诈方式颁发的证书（例如，你正在运营一个高知名度的网站）受到攻击，那么钉扎是保护自己的唯一方式。对于其他人，钉扎可能并不值得。

10.3.4　在本机应用程序中使用钉扎

最直接使用钉扎的是本机应用程序，你可以控制通信的双方。这正是桌面应用和移动应用程序的情况。在一个日益互联的世界，越来越多的应用程序有它们交互的后端，并且很多都使用HTTPS进行通信。

1. 私有后端

有两种方法可以采用。当后端仅被你自己的应用程序使用的时候，你可以生成自己的根证书私钥，并用它来颁发自己的证书。通过将根证书公钥颁发到你的应用程序，就能可靠地验证证书签名。

在许多平台上，这种类型的钉扎很容易做到。例如，Java附带了一些默认使用的可信根。每当你打开一个到网站的HTTPS连接，这些可信根可用于验证连接的真实性。但是，因为你不想相信所有的根，可以创建自己的*可信证书库*（trust store），然后只在里面配置你自己的根。无论何时你通过指定自己的可信证书库来建立到自己网站的HTTPS连接，那么就是在使用钉扎。

如果不希望维护自己的根密钥，如前所述，可以用SPKI钉扎。如果你已经写了一些代码，Moxie Marlinspike在他的文章中介绍了这两种方法。[①]

---

① Your app shouldn't suffer SSL's problems，http://www.thoughtcrime.org/blog/authenticity-is-broken-in-ssl-but-your-app-ha/（Moxie Marlinspike，2011年12月5日）。

Android从4.2版本开始，已经比较有限地支持了公钥钉扎。[①]

2. 公共后端

在某些情况下，应用程序使用有第三方（即公共）访问的后端，于是必须从公共CA获取证书。这样一来，其他人将能够连接到服务并且验证其真实性。你将无法部署钉扎来确保它们的访问安全，直到至少有一个钉扎议案得到广泛支持。

如果你还是想保护自己的应用程序访问，可以遵循10.3.3节中的建议和对公钥钉扎的建议。一个更安全的方法可能是创建另一个私有后端，在这种情况下，你可以用你自己的根密钥证书。

10.3.5　Chrome 公钥钉扎

Google在Chrome 12中开始尝试公钥钉扎，[②]配备了可自定义的HSTS和钉扎设置的用户界面。[③]之后，在Chrome13中增加（预加载）了大部分他们自己的网站钉扎。[④]

幕后HSTS预加载和钉扎使用相同的机制；所需要的信息在浏览器本身中进行硬编码。由于Chrome是基于开源的Chromium浏览器，我们可以查看包含此信息的源文件。[⑤]

这里只有一个策略文件，它包含有两个清单的一个JSON结构：(1) 支持HSTS或钉扎的网站；(2) 用于为它们定义可接受的公钥的pinset。

每个网站条目都有相应的HSTS配置和期望的pinset信息：

```
{ "name": "encrypted.google.com",
  "include_subdomains": true,
  "mode": "force-https",
  "pins": "google"
}
```

pinset是一个允许的SPKI散列值的集合；它使用的名称并不是证书文件中的，而是浏览器附带的：

```
{ "name": "google",
  "static_spki_hashes": [
      "GoogleBackup2048",
      "GoogleG2"
  ]
}
```

通过pinset方式，Chrome会创建一个公钥的白名单用于证书链钉扎的网站。这种格式也允许公钥的黑名单（通过`bad_static_spki_hashes`参数设置），但目前还没有网站使用。还有一个规

① Certificate pinning in Android 4.2，http://nelenkov.blogspot.ca/2012/12/certificate-pinning-in-android-42.html（Nikolay Elenkov，2012年12月12日）。

② New Chromium security features, June 2011，http://blog.chromium.org/2011/06/new-chromium-security-features-june.html（The Chromium Blog，2011年6月14日）。

③ Chrome浏览器目前版本仍包含此用户界面，可以通过chrome://net-internals/#hsts访问。

④ Public key pinning，https://www.imperialviolet.org/2011/05/04/pinning.html（Adam Langley，2011年5月4日）。

⑤ `transport_security_state_static.json`， https://chromium.googlesource.com/chromium/src/net/+/master/http/transport_security_state_static.json（Chromium源代码，检索于2015年4月20日）。

定是，当SNI不可用的时候禁止钉扎，这对于一些仅当启用SNI时才提供正确证书链的网站是必要的。[①]

正如你所看到的，这一切似乎很简单。由于Chrome的开发者已经慷慨地允许其他人在其浏览器中包含他们的钉扎信息，一些高知名度的网站和项目（例如Twitter和TOR）也受钉扎保护。数以百计的网站HSTS信息被预加载。

> **警告**
> 为了允许用户MITM自己的流量，钉扎对手动添加的根证书不作限制。一方面允许防病毒产品进行本地调试（例如，使用本地开发者代理）和内容检查；另一方面，它也允许透明的企业通信拦截。据报道，一些恶意软件作者安装自定义证书进行MITM攻击；此类证书也将绕过钉扎的验证。[②]

Chrome浏览器包含将钉扎验证失败信息报告给Google的报告机制。（有趣的是，出于隐私的原因，该报告只在Google自己的属性中启用）。我们知道这些是因为Chrome的钉扎发现了一些PKI事件：DigiNotar、TURKTRUST和ANSSI。你可以在第4章中了解这方面的内容。

> **注意**
> 2014年9月发布的Firefox 32增加了对硬编码公钥钉扎的支持，这类似于已经在Chrome浏览器中使用的机制。[③]Firefox 34增加了在Android上对硬编码公钥钉扎的支持。

10.3.6 Microsoft Enhanced Mitigation Experience Toolkit

Internet Explorer目前不支持网站自主的钉扎，但Microsoft提供了一个附加的名为enhanced mitigation experience toolkit（EMET）[④]的工具，可用于最终用户单独保护自己。虽然EMET大多集中在缓冲溢出和类似的攻击，其特色之一是证书钉扎。EMET5中包含了几个关键Microsoft网站、Facebook、Twitter和Yahoo的钉扎规则。用户如果想用，可以添加自己的钉扎。[⑤]

10.3.7 HTTP 公钥钉扎扩展

HTTP公钥钉扎扩展（public key pinningextension for HTTP，HPKP）[⑥]自2011年开始开发，是

① Chrome支持SNI，这就是为什么这个功能初看起来似乎不合逻辑。但是，对于某些不支持扩展（这意味着Chrome浏览器无法发送SNI信息）的情况，Chrome已准备好从TLS 1.2回退到SSL 3。

② New Man-in-the-Middle attacks leveraging rogue DNS，http://blog.phishlabs.com/new-man-in-the-middle-attacks-leveraging-rogue-dns（Don Jackson，PhishLabs，2014年3月26日）。

③ Public key pinning released in Firefox，https://blog.mozilla.org/security/2014/09/02/public-key-pinning/（Mozilla Security Blog，2014年9月2日）。

④ The Enhanced Mitigation Experience Toolkit，https://support.microsoft.com/en-us/kb/2458544（Microsoft，2013年2月12日）。

⑤ Announcing EMET 5.0 Technical Preview，http://blogs.technet.com/b/srd/archive/2014/02/25/announcing-emet-5-0-technical-preview.aspx（Microsoft Security Research and Defense Blog，2014年2月25日）。

⑥ RFC 7469: Public Key Pinning Extension for HTTP，https://datatracker.ietf.org/doc/rfc7469/（Evans等，2015年4月）。

对HTTP用户代理的公钥钉扎规范。它由Google发起，虽然已经在Chrome中实现钉扎，但也清楚手动维护钉扎站点列表很难规模化。在写这本书的时候，HPKP已接近完成。虽然几乎没有浏览器厂商声明对它们的支持，但Chrome和Firefox期望一旦HPKP完成就在自己的浏览器中实现它。

因为HPKP和HSTS之间有许多相似之处，如果你还没有阅读HSTS部分（本章前面），我建议你现在就去读一下。以下是这些共同点的快速概览。

- HPKP设置为HTTP级别，使用`Public-Key-Pins`（PKP）响应头。
- 策略通过`max-age`参数设置保留期，它指定以秒为单位的持续时间。
- 使用`includeSubDomains`参数时钉扎可以扩展到子域。
- PKP响应头只能用于没有任何错误的安全加密；如果出现多个响应头，仅会处理第1个。
- 接收到新的PKP头时，其中的信息会覆盖先前保存的钉扎和元数据。

钉扎是通过指定散列算法和使用该算法计算的SPKI指纹来创建的。例如：

```
Public-Key-Pins: max-age=2592000;
       pin-sha256="E9CZ9INDbd+2eRQozYqqbQ2yXLVKB9+xcprMF+44U1g=";
       pin-sha256="LPJNul+wow4m6DsqxbninhsWHlwfp0JecwQzYp0LmCQ="
```

目前唯一支持的散列算法是SHA256；配置钉扎时使用`sha256`标识符。指纹是使用Base64编码进行编码的。

为了启用钉扎，你必须指定策略保留期限，并提供至少两个钉扎。其中一个钉扎必须存在于收到钉扎的连接的证书链中。其他钉扎则不能存在。由于钉扎是一个潜在的危险操作（很容易出错，并执行自我造成的拒绝服务攻击），第二个钉扎需要作为备份。推荐的做法是，从不同的CA备份证书并且离线保存。此外，建议对备份证书偶尔进行测试。你肯定不希望在需要它的时候才发现它已经无效了。

1. 报告

HPKP不同于HSTS，而是与CSP类似。HPKP指定一种机制用于用户代理报告钉扎验证失败。该功能通过`report-uri`参数启用，它包含提交报告的入口。

```
Public-Key-Pins: max-age=2592000;
       pin-sha256="E9CZ9INDbd+2eRQozYqqbQ2yXLVKB9+xcprMF+44U1g=";
       pin-sha256="LPJNul+wow4m6DsqxbninhsWHlwfp0JecwQzYp0LmCQ=";
       report-uri=http://example.com/pkp-report
```

报告使用POST HTTP请求提交，其中包括一个JSON结构在请求主体。例如（取自HPKP规范的草案）：

```
{
  "date-time": "2014-04-06T13:00:50Z",
  "hostname": "www.example.com",
  "port": 443,
  "effective-expiration-date": "2014-05-01T12:40:50Z"
  "include-subdomains": false,
  "served-certificate-chain": [
    "-----BEGIN CERTIFICATE-----\n
    MIIEBDCCAuygAwIBAgIDAjppMA0GCSqGSIb3DQEBBQUAMEIxCzAJBgNVBAYTAlVT\n
```

```
    ...
    HFa9llF7b1cq26KqltyMdMKVvvBulRP/F/A8rLIQjcxz++iPAsbw+zOzlTvjwsto\n
    WHPbqCRiOwY1nQ2pM714A5AuTHhdUDqB1O6gyHA43LL5Z/qHQF1hwFGPa4NrzQU6\n
    yuGnBXj8ytqUOCwIPX4WecigUCAkVDNx\n
    -----END CERTIFICATE-----",
    ...
  ],
  "validated-certificate-chain": [
    "-----BEGIN CERTIFICATE-----\n
    MIIEBDCCAuygAwIBAgIDAjppMA0GCSqGSIb3DQEBBQUAMEIxCzAJBgNVBAYTAlVT\n
    ...
    HFa9llF7b1cq26KqltyMdMKVvvBulRP/F/A8rLIQjcxz++iPAsbw+zOzlTvjwsto\n
    WHPbqCRiOwY1nQ2pM714A5AuTHhdUDqB1O6gyHA43LL5Z/qHQF1hwFGPa4NrzQU6\n
    yuGnBXj8ytqUOCwIPX4WecigUCAkVDNx\n
    -----END CERTIFICATE-----",
    ...
  ],
  "known-pins": [
    "pin-sha256=\"d6qzRu9zOECb90Uez27xWltNsjOe1Md7GkYYkVoZWmM=\"",
    "pin-sha256=\"E9CZ9INDbd+2eRQozYqqbQ2yXLVKB9+xcprMF+44U1g=\""
  ]
}
```

2. 部署非强制模式

当HPKP部署非强制模式时报告功能特别有用，可以通过使用Public-Key-Pins-Report-Only响应头实现。这种方法允许组织部署钉扎而不担心失败，确保其配置正确之后才切换成强制模式。根据自己的风险，某些组织可能会选择永不启用强制模式；知道被攻击通常像避免被攻击一样有用。

10.3.8　DANE

已命名实体的基于DNS的身份验证（DNS-based authentication of named entities，DANE）[①]是设计用于在域名与一个或多个加密标识之间提供关联的一个标准。想法是域名的所有者已经控制自己的DNS配置，可以使用DNS作为一个独立的信道来分配TLS强身份验证需要的信息。DANE直截了当，并且相对容易部署，但本身不提供任何安全性。相反，它依赖域名系统安全扩展（domain name system security extensions，DNSSEC）的可用性。[②]

DNSSEC是扩展不提供任何安全性的当前DNS实现的一次尝试，此次扩展使用支持使用数字签名进行身份验证的新架构。通过身份验证，我们能够验证获取的DNS信息是否正确。DNSSEC颇有争议，它已经发展超过十年，但部署进展缓慢。对于DNSSEC是对当前DNS系统的改进，还是应该寻求替代，专家的意见也是大相径庭。

今天通常可以在后端部署DNSSEC，但更多最终用户的支持（在客户端操作系统中）将要花

① RFC 6698: The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA，https://tools.ietf.org/html/rfc6698（Hoffman和Schlyter，2012年8月）。

② Domain Name System Security Extensions，https://en.wikipedia.org/wiki/Domain_Name_System_Security_Extensions（维基百科，检索于2014年6月29日）。

费很长的时间。

1. DANE使用案例

在当前的TLS验证模型中，我们依靠的方法有两个步骤：(1)有一组权威的证书颁发机构，我们信任他们只对真正的域名所有者颁发证书；(2) 当一个网站被访问时，用户代理（如浏览器）检查证书和预期的名称是否正确。需要这种拆分模型，因为对遥远的当事人进行身份验证（例如，从来没有见过的人）得到正确结果是非常困难的，尤其是规模化时。该系统的设计是建立在以下假设之上：由DNS提供的信息是不可靠的（也就是说，主动网络攻击者可以暗中破坏它）。

通过DNSSEC，可以确保我们收到的信息来自域名拥有者的通信信道；这意味着，我们并不一定需要第三方（CA）为其提供任何担保。下面是几个有趣的用例。

- 安全部署的自签名证书

  现在自签名证书被认为是不安全的，因为普通用户没有办法区分真正的证书和自签名的MITM证书。换句话说，所有的自签名证书看起来是一样的。但是，我们可以使用一个安全的DNS来钉扎证书，从而使我们的用户代理知道自己使用的证书是正确的。MITM证书很容易被发现。

- 安全部署私有根证书

  如果你能安全钉扎服务器证书，那么一样也可以钉扎证书链中的其他证书。这意味着，你可以创建自己的根证书，并让用户代理信任它，但只针对你自己的网站。这是前面用例的变化，那些拥有多个站点的人会对此很感兴趣。你只需创建一次所有网站的根证书的钉扎，而不需要钉扎单独的证书（其中有许多人需要经常旋转证书）。

- 证书和公钥钉扎

  DANE不一定要取代当前的信任体系。你可以容易地钉扎CA颁发的证书和公共CA的根。通过这样做可以减少攻击面，有效地决定允许由哪些CA颁发证书。

2. 实现

DANE引入了一个新的DNS条目类型，称为TLSA资源记录（TLSA resource record，TLSA RR或TLSA），其用于承载证书关联。TLSA包括四个字段：(1) 证书用途（certificate usage）用于指定一个证书链中的哪部分被钉扎，以及如何进行验证；(2) 选择器（selector）指定什么元素用于钉扎；(3) 匹配类型（matching type）用以选择精确匹配或散列；(4) 证书关联数据（certificate association data），它带有真正用于匹配的原始数据。这四个字段的不同组合用于部署不同类型的钉扎。

- 证书用途

证书用途（certificate usage）字段可以有四种不同的值。在一开始的RFC中，它的值只是简单的数字从0到3。后续RFC增加了缩写使其更容易记忆。[①]

① RFC 7218: Adding Acronyms to Simplify Conversations about DANE，https://tools.ietf.org/html/rfc7218（Gudmundsson，2014年4月）。

- ❑ CA约束（0；PKIX-TA）
  创建一个CA的钉扎，其匹配的证书必须在链中的任何地方存在。PKIX像往常一样执行验证，并且根证书必须来自一个可信CA。
- ❑ 服务证书约束（1；PKIX-EE）
  创建一个最终实体钉扎，其证书必须在证书链的第一个位置呈现。PKIX像往常一样执行验证，并且根证书必须来自一个可信CA。
- ❑ 信任锚断言（2；DANE-TA）
  对必须存在于信任链中的CA证书（根或中间）创建信任锚钉扎。PKIX像往常一样执行验证，但用户代理必须信任被钉扎的CA证书。此选项可用于不通过公共CA颁发的证书。
- ❑ 域颁发的证书（3；DANE-EE）
  创建一个最终实体钉扎，其证书必须在证书链的第一个位置呈现。没有PKIX验证，并且假定被钉扎的证书是可信的。

- 选择器

选择器（selector）字段指定关联如何呈现。这使我们能够创建一个与证书（0；Cert）的关联或者创建一个与SubjectPublicKeyInfo字段（1；SPKI1）的关联。

- 匹配类型

匹配类型（matching type）字段指定匹配是通过直接比较（0；Full）或通过散列（分别是1和2，或SHA2-256和SHA2-512）。支持SHA256是必需的；建议支持SHA512。

- 证书关联数据

证书关联数据（certificate association data）字段包含用于关联的原始数据。其内容通过在TLSA记录其他三个字段的值来确定。证书始终是一个关系的出发点，假定是DER格式。

3. 部署

抛开DNSSEC配置和签名（只因为它超出了本书的范围），DANE很容易部署。你需要做的只是在正确的名称下添加一条新的TLSA纪录。这个名称不仅是想要稳固的域名，它是由点号分隔的以下三段组合。

- ❑ 第一段是该服务使用的端口，带有前缀下划线。例如，HTTPS用_443，SMTP用_25。
- ❑ 第二段是加上前缀下划线的协议。支持三种协议：UDP、TCP和SCTP。对于HTTPS，该段将为_tcp。
- ❑ 第三部分是你想要为其创建关联的完整限定域名。例如，www.example.com。

在下面的例子中，关联是在域名与CA的公钥之间建立的（证书用途为0），由SubjectPublicKeyInfo字段（选择器为1）通过其十六进制编码的SHA256散列（匹配类型为1）进行标识：

```
_443._tcp.www.example.com. IN TLSA (
     0 1 1 d2abde240d7cd3ee6b4b28c54df034b9
           7983a1d16e8a410e4561cb106618e971 )
```

DANE通过为所需的域名增加一条或多条TLSA记录来激活。如果至少有一个关联存在，用户代理必须建立一个匹配；否则，它们必须中止TLS握手。如果没有关联，则用户代理可以按照正常情况处理TLS连接。

因为可以为一个域名配置多个关联（TLSA记录），有可能存在一个或多个备份关联。没有任何停机的轮转关联是可能的。与HPKP不同，DANE不指定记忆效应，但本身有一个内置的DNS：生存时间（time to live，TTL）值，该值是记录可被缓存的持续时间。然而，由于缺乏明确的记忆效应是DANE的优势；错误很容易通过重新配置DNS纠正。在部署时，特别是刚开始时，最好用尽可能短的TTL。

一个潜在的缺点是DANE RFC并没有指定匹配的关联无法找到时的用户交互。例如，HPKP建议用户在钉扎失败的情况下可以手动中断钉扎。这是一把双刃剑：在遇到真正攻击的情况下固执的用户可能会终止覆盖安全机制；另一方面，使用DANE配置错误时没有退路。另一个问题是DANE不支持报告，这使得很难找到关联的匹配失败是否发生过。

4. 应用支持

在写这篇文章的时候，DANE还不被主流的浏览器所支持。增加对它的支持是困难的，因为DANE建立在DNSSEC之上；直到操作系统开始使用DNSSEC之后，浏览器还需要自己实现DNSSEC解析。Chrome早在2011（Chrome14）就尝试了DANE，但最终删除了对它支持，理由是缺乏使用。[①]因为这一点，当前对DANE唯一感兴趣是发烧友和那些想要学习公共TLS身份验证的相关人员。

尽管缺乏支持，但今天我们仍可以试用DANE，这多亏了DNSSEC TLSA Validator插件，它适用于所有主流浏览器。[②]如果成功安装插件，你可以测试VeriSign运营的一个示范网站。[③]

10.3.9　证书密钥可信保证

证书密钥可信保证（trust assertions for certificate key，TACK）[④]是公钥钉扎的建议，其目的是独立于公共CA和DNS。想法是网站经营者建立自己的签名密钥（也称为TACK签名密钥，TACK signing key或TSK），为独立提供支援。一旦用户代理识别一个特定网站的TSK，该密钥可用于撤销旧服务器密钥，颁发新密钥，等等。换言之，TSK类似于私有CA。虽然建议每个站点一个TSK，但相关网站可以依赖相同的签名密钥。

TACK是全部钉扎建议里最雄心勃勃的，而且也是最复杂的。遵循它的用户代理在它的`ClientHello`里提交一个空的`tack`扩展表示支持TACK。作为回应，一个遵循的服务器使用相同的扩展发送一个或多个钉扎，这些钉扎是用该网站的TSK对服务器公钥签名的。钉扎出现在第一次，

---

① DNSSEC authenticated HTTPS in Chrome，https://www.imperialviolet.org/2011/06/16/dnssecchrome.html（Adam Langley，2011年6月16日）。

② DNSSEC/TLSA Validator add-on for Web Browsers，https://www.dnssec-validator.cz/（CZ.NIC，检索于2015年7月）。

③ Verisign Labs DANE Demonstration，http://dane.verisignlabs.com/（VeriSign，检索于2015年7月）。

④ Trust Assertions for Certificate Keys，https://tools.ietf.org/html/draft-perrin-tls-tack-02（Marlinspike和Perrin，2013年1月）。

但只有第二次才被激活。没有固定的策略保存时间。相反，在每次访问时，用户代理就会通过从当前时间戳减去钉扎第一次出现的时间戳来计算出新的策略保留时间。还存在一个30天的最大限制。

TACK是有趣的，因为它可以与任何协议一起使用（与仅用于HTTP的HPKP不同）。另一方面，使用单独的签名密钥引入了更多的复杂性。此外，它需要改变TLS协议。在当前这个时候，尚不清楚浏览器厂商是否有支持它的计划。

10.3.10　证书颁发机构授权

证书颁发机构授权（certification authority authorization，CAA）[①]提出了一种域名所有者授权CA为自己的域名签发证书的方法。其目的是深度防御证书颁发在验证过程中的攻击；通过CAA，CA可以肯定自己与真正的域名所有者进行通信。

CAA依赖于DNS进行策略分发；它建议使用DNSSEC但不强制。它通过增加*CAA资源记录*（CAA resource record，CAA RR）来扩展DNS，用于创建授权条目。

CAA支持多个*属性标记*（property tag），它们是用于CA的指令。例如，`issue`标记可用于允许CA（由其域名标识）为某个特定域名颁发证书：

```
certs.example.com          CAA 0 issue "ca.example.net"
```

同样的标签可以用来禁止证书颁发：

```
nocerts.example.com        CAA 0 issue ";"
```

其他标记还包括`issuewild`（涉及泛域名证书）和`iodef`，后者定义了一个通信信道（例如，电子邮件地址），用于CA向网站所有者报告无效证书颁发申请。

CAA真正成功需要由CA广泛采用。攻击者可以随时针对不遵循规范的CA，并从中获得假证书。当然，从遵循的CA角度来说，这并不一定是失败；任何降低攻击可能性的事务都被视为正面的。但是，如果没有足够的CA支持此功能，网站所有者不可能作出努力来配置对他们的属性授权。

与DANE一样，CAA最适合DNSSEC。若无DNSSEC，CA必须特别注意不要将自己暴露给DNS欺骗攻击。

10

① RFC 6944: DNS Certification Authority Authorization (CAA) Resource Record，https://datatracker.ietf.org/doc/rfc6844/（Hallam-Baker，2013年1月）。

第 11 章 OpenSSL

OpenSSL是一个开源项目，包括密码库和SSL/TLS工具集。从项目的官方站点可以看到：

> OpenSSL项目是**安全套接字层**（secure sockets layer，SSL）和**传输层安全**（transport layer security，TLS）协议的一个实现，是大家共同努力开发出的代码可靠、功能齐全、商业级别的开源工具集。项目由遍布世界的志愿者所组成的社区进行管理，他们通过互联网进行沟通、计划和开发OpenSSL工具集以及相关的文档。

OpenSSL在这一领域已经成为事实上的标准，并且拥有比较长的历史。OpenSSL的代码前身是SSLeay[①]，由Eric A. Young和Tim J. Hudson在1995年开发出来。因为后来Eric和Tim停止了SSLeay的更新，转而为澳大利亚的RSA公司开发商业版本的SSL/TLS工具集BSAFE SSL-C，所以OpenSSL项目在1998年的最后几天诞生了。

现在几乎所有的服务器软件和很多客户端软件都在使用OpenSSL，其中基于命令行的工具是进行密钥、证书管理以及测试最常用到的软件了。有意思的是，之前很多使用其他库作为SSL/TLS解析的浏览器正发生一些变化，例如Google正在将Chrome迁移到他们自己的OpenSSL分支BoringSSL。[②]

OpenSSL是基于OpenSSL和SSLeay双许可证下的授权模式，两种许可都类似于BSD，同时有一个建议的条款。OpenSSL的许可在很长一段时间里都是争论的焦点，因为其中任何一个许可都与GPL家族不兼容。因此，你会发现那些基于GPL许可的程序比较偏向于使用GnuTLS。

11.1 入门

如果你使用的操作系统是基于Unix的，那么几乎可以肯定上面已经安装了OpenSSL，我们的入门步骤也会变得非常简单。唯一可能的问题是版本不是最新的。在本节中，我假设你已经在使用Unix平台，因为OpenSSL天生就基于Unix平台。

Windows的用户就稍微麻烦了，最简单的情况是你仅仅需要使用OpenSSL的命令行工具，可

① 名称SSLeay中的eay是Eric A. Young的首字母缩写。

② BoringSSL，https://www.chromium.org/Home/chromium-security/boringssl（Chromium，检索于2015年6月30日）。

以在OpenSSL的网站上找到Shining Light Production的链接[①]，它拥有已经编译好的Windows版本的OpenSSL。如果你想要OpenSSL的所有功能，需要确保那些编译好的OpenSSL都是基于同一个版本编译出来的，否则可能会遇到难以解决的崩溃问题。最好的办法是使用一个包含你需要的所有程序的软件包。例如，如果你想要在Windows上运行Apache，那么可以直接从Apache Lounge获取对应的二进制文件。[②]

11.1.1　确定 OpenSSL 版本和配置

在开始之前，你要首先确定当前使用的OpenSSL版本。下面是我在Ubuntu 12.04 LTS上面运行`openssl vesion`获得的版本信息，后面所有的演示都是基于这个环境：

```
$ openssl version
OpenSSL 1.0.1 14 Mar 2012
```

在写这本书的时候，OpenSSL 0.9.x正在向OpenSSL 1.0.x发展。1.0.1最大的意义在于它是第一个同时支持TLS 1.1和1.2的版本。总体来说，支持新协议是大趋势，所以我们还要经历一段新老协议不互通的过程。

> **注意**
> 不同的操作系统经常会修改OpenSSL的代码，主要是修复一些已知漏洞。然而项目的名称和版本号经常保持原样，也没有任何迹象表明代码其实是原项目的一个分支，一些行为已经变化了。例如现在使用的Ubuntu 12.04 LTS的OpenSSL是基于1.0.1c版本，全名是`openssl 1.0.1-4ubuntu5.16`，包含了很多已知问题的补丁。[③]

可以使用`-a`开关获取完整的版本信息：

```
$ openssl version -a
OpenSSL 1.0.1 14 Mar 2012
built on: Fri Jun 20 18:54:15 UTC 2014
platform: debian-amd64
options: bn(64,64) rc4(8x,int) des(idx,cisc,16,int) blowfish(idx)
compiler: cc -fPIC -DOPENSSL_PIC -DZLIB -DOPENSSL_THREADS -D_REENTRANT -DDSO_DLFCN↩
-DHAVE_DLFCN_H -m64 -DL_ENDIAN -DTERMIO -g -O2 -fstack-protector↩
--param=ssp-buffer-size=4 -Wformat -Wformat-security -Werror=format-security -D↩
_FORTIFY_SOURCE=2 -Wl,-Bsymbolic-functions -Wl,-z,relro -Wa,--noexecstack -Wall↩
-DOPENSSL_NO_TLS1_2_CLIENT -DOPENSSL_MAX_TLS1_2_CIPHER_LENGTH=50 -DMD32_REG_T=int↩
-DOPENSSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_MONT5 -DOPENSSL_BN_ASM↩
_GF2m -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DMD5_ASM -DAES_ASM -DVPAES_ASM -DBSAES↩
_ASM -DWHIRLPOOL_ASM -DGHASH_ASM
OPENSSLDIR: "/usr/lib/ssl"
```

① Win32 OpenSSL，http://slproweb.com/products/Win32OpenSSL.html（Shining Light Productions，检索于2014年7月3日）。

② Apache 2.4 VC14 Binaries and Modules，http://www.apachelounge.com/download/（Apache Lounge，检索于2015年7月15日）。

③ Precise版本的OpenSSL源代码包，https://launchpad.net/ubuntu/precise/+source/openssl（Ubuntu，检索于2014年7月3日）。

上面最后一行的输出（/usr/lib/ssl）是OpenSSL默认情况下查找配置和证书的目录。在我的系统里，该位置是/etc/ssl的别名（也就是软链接），Ubuntu会在其中保存与TLS相关的文件：

```
lrwxrwxrwx 1 root root   14 Apr 19 09:28 certs -> /etc/ssl/certs
drwxr-xr-x 2 root root 4096 May 28 06:04 misc
lrwxrwxrwx 1 root root   20 May 22 17:07 openssl.cnf -> /etc/ssl/openssl.cnf
lrwxrwxrwx 1 root root   16 Apr 19 09:28 private -> /etc/ssl/private
```

misc/目录包含一些补充脚本，其中最有用的脚本允许你实现一个私有的证书颁发机构。

11.1.2　构建 OpenSSL

大多数情况下使用操作系统默认提供的OpenSSL就够了，但最好还是升级到最新版本。例如你当前的系统还在使用OpenSSL 0.9.x，而你想使用更新的TLS协议（在OpenSSL 1.0.1以上版本中才有）。当然，新版本的OpenSSL可能无法提供你想要的所有功能，例如在Ubuntu 12.04 LTS上的openssl s_client命令就不支持SSL 2。虽然默认不支持SSL 2是对的，但是如果需要测试别的服务器是否支持SSL 2，我们就需要这类功能。

你可以先下载最新版本的OpenSSL（我使用的是1.0.1h）：

```
$ wget http://www.openssl.org/source/openssl-1.0.1h.tar.gz
```

在编译之前我们还需要进行配置。一般情况下，我们配置的安装目录不要与系统默认提供的OpenSSL目录一样，例如：

```
$ ./config \
--prefix=/opt/openssl \
--openssldir=/opt/openssl \
enable-ec_nistp_64_gcc_128
```

enable-ec_nistp_64_gcc_128参数可以让我们使用优化后的一些常用的椭圆曲线算法，这个优化基于编译器的一些特性，默认情况下会关闭这些特性，而且无法自动检测。

然后执行下面这些命令：

```
$ make depend
$ make
$ sudo make install
```

然后在/opt/openssl目录下面，我们可以看到：

```
drwxr-xr-x 2 root root  4096  Jun 3 08:49 bin
drwxr-xr-x 2 root root  4096  Jun 3 08:49 certs
drwxr-xr-x 3 root root  4096  Jun 3 08:49 include
drwxr-xr-x 4 root root  4096  Jun 3 08:49 lib
drwxr-xr-x 6 root root  4096  Jun 3 08:48 man
drwxr-xr-x 2 root root  4096  Jun 3 08:49 misc
-rw-r--r-- 1 root root 10835 Jun 3 08:49 openssl.cnf
drwxr-xr-x 2 root root  4096  Jun 3 08:49 private
```

private/目录默认是空的，因为我们还没有生成任何私钥，certs目录也是如此。OpenSSL不包括任何根证书，因为维护一个可信证书库不在OpenSSL项目的范围之内。幸运的是，很多操作系

统可能已经安装了可信证书库，当然你也可以自己创建一个，11.1.4节会具体介绍。

> **注意**
> 编译软件之前，熟悉编译器的默认配置是很重要的。系统提供的软件一般都会尽可能地使用到编译器的优化选项，但是如果你自己编译的话，就不一定能保证这些优化选项都能用到。[①]

11.1.3　查看可用命令

OpenSSL包含了很多密码相关的工具。我计算了一下，我的版本大概有46个，简直可以称为密码学领域的瑞士军刀。即便现在你只会用到其中的一部分工具，但是还是有必要熟悉所有的工具，这样在将来需要的时候才能知道你有哪些工具可以使用。

OpenSSL没有专门的help关键字，任何时候输入OpenSSL无法识别的命令，就会显示帮助文本：

```
$ openssl help
openssl:Error: 'help' is an invalid command.

Standard commands
asn1parse         ca               ciphers          cms
crl               crl2pkcs7        dgst             dh
dhparam           dsa              dsaparam         ec
ecparam           enc              engine           errstr
gendh             gendsa           genpkey          genrsa
nseq              ocsp             passwd           pkcs12
pkcs7             pkcs8            pkey             pkeyparam
pkeyutl           prime            rand             req
rsa               rsautl           s_client         s_server
s_time            sess_id          smime            speed
spkac             srp              ts               verify
version           x509
```

第一部分帮助列出了所有可以使用的工具。如果对于某个命令，想获取更加详细的信息，可以使用man加上工具的名称。例如man ciphers会告诉我们密码套件是如何配置的。

帮助信息不止上面这些，但是剩下的就没这么有意思了，在第二部分的帮助信息里我们可以看到可用的消息摘要命令：

```
Message Digest commands (see the `dgst' command for more details)
md4              md5              rmd160           sha
sha1
```

然后在第三部分的帮助信息中，我们可以看到所有的加密命令：

```
Cipher commands (see the `enc' command for more details)
aes-128-cbc      aes-128-ecb      aes-192-cbc      aes-192-ecb
aes-256-cbc      aes-256-ecb      base64           bf
```

① compiler hardening in Ubuntu and Debian，https://outflux.net/blog/archives/2014/02/03/compiler-hardening-in-ubuntu-and-debian/（Kees Cook，2014年2月3日）。

```
bf-cbc             bf-cfb             bf-ecb             bf-ofb
camellia-128-cbc   camellia-128-ecb   camellia-192-cbc   camellia-192-ecb
camellia-256-cbc   camellia-256-ecb   cast               cast-cbc
cast5-cbc          cast5-cfb          cast5-ecb          cast5-ofb
des                des-cbc            des-cfb            des-ecb
des-ede            des-ede-cbc        des-ede-cfb        des-ede-ofb
des-ede3           des-ede3-cbc       des-ede3-cfb       des-ede3-ofb
des-ofb            des3               desx               rc2
rc2-40-cbc         rc2-64-cbc         rc2-cbc            rc2-cfb
rc2-ecb            rc2-ofb            rc4                rc4-40
seed               seed-cbc           seed-cfb           seed-ecb
seed-ofb           zlib
```

11.1.4　创建可信证书库

OpenSSL没有自带可信根证书（也叫作可信证书库），所以如果你是自己从头开始安装的话，那么就需要从别的地方找找了。一种选择是使用操作系统自带的可信证书库，一般来说没有问题，但是这个可信证书库可能不是最新的。更好的一种选择是从Mozilla那里获取，虽然麻烦一点，但是Mozilla花费了很多时间维护一个可靠的可信证书库，例如我为自己在SSL Labs写的评估工具使用的就是这个可信证书库。

因为是开源的，所以Mozilla将可信证书库保存在源代码存储库中：

```
https://hg.mozilla.org/mozilla-central/raw-file/tip/security/nss/lib/ckfw/builtins/certdata.txt
```

但是Mozilla将这些证书以一种比较特殊的格式存放，其他人很少使用。所以如果不介意的话，我们可以从第三方（例如Curl项目）去获取最新的PEM（privacy-enhanced mail）格式的可信证书库，这种格式可以直接使用：

```
http://curl.haxx.se/docs/caextract.html
```

当然，如果你更愿意直接从Mozilla下载可信证书库，也有现成的用Perl或者Go语言编写的脚本，这样你就无需自己编写了。在后面几节我会介绍这两个工具。

> **注意**
> 如果你最终决定自己编写解析Mozilla可信证书库脚本的话，需要注意根证书文件中实际包含了两种类型的证书：一部分是可信的，另外一部分是明确不可信的。Mozilla使用这种方式去禁用那些被盗用的中间证书（例如DigiNotar的那些老证书），上面提到的Perl和Go脚本会识别出不可信证书。

1. 使用Perl脚本

Curl项目提供了由Guenter Knauf使用Perl编写的Mozilla可信证书库转换脚本：

```
https://raw.github.com/bagder/curl/master/lib/mk-ca-bundle.pl
```

下载这个脚本之后直接运行就会从Mozilla下载最新的证书数据并且转换为PEM格式：

```
$ ./mk-ca-bundle.pl
Downloading 'certdata.txt' ...
```

```
Processing 'certdata.txt' ...
Done (156 CA certs processed, 19 untrusted skipped).
```

如果还有之前下载过的证书数据，这个脚本会将两份证书数据进行对比，然后只处理更新的部分。

2. 使用Go脚本

如果你更喜欢使用Go语言，那么可以从GitHub下载Adam Langley编写的工具：

```
https://github.com/agl/extract-nss-root-certs
```

开始转换之前先下载这个小工具：

```
$ wget https://raw.github.com/agl/extract-nss-root-certs/master/convert_mozilla_certdata.go
```

然后下载Mozilla的证书数据：

```
$ wget https://hg.mozilla.org/mozilla-central/raw-file/tip/security/nss/lib/ckfw/builtins/certdata. ↩
txt --output-document certdata.txt
```

最后使用下面的命令转换文件：

```
$ go run convert_mozilla_certdata.go > ca-certificates
2012/06/04 09:52:29 Failed to parse certificate starting on line 23068: negative serial number
```

在我的例子里面有一个无效的证书是Go X.509无法处理的，其他的都正常处理了。

11.2 密钥和证书管理

大多数用户借助OpenSSL是因为希望配置并运行能够支持SSL的Web服务器。整个过程包括3个步骤：(1) 生成强加密的私钥；(2) 创建证书签名申请（certificate signing request，CSR）并且发送给CA；(3) 在你的Web服务器上安装CA提供的证书。这些步骤（还有其他一些）会在本节详细说明。

11

11.2.1 生成密钥

在使用公钥加密之前的第一步是生成一个私钥。在开始之前，你必须进行几个选择。

- 密钥算法

  OpenSSL支持RSA、DSA和ECDSA密钥，但是在实际使用场景中不是所有密钥类型都适用的。例如对于Web服务器的密钥，所有人都使用RSA，因为DSA一般因为效率问题会限制在1024位（Internet Explorer不支持更长的DSA密钥），而ECDSA还没有被大部分的CA支持。对于SSH来说，一般都是使用DSA和RSA，而不是所有的客户端都支持ECDSA算法。

- 密钥长度

  默认的密钥长度一般都不够安全，所以我们需要指定要配置的密钥长度。例如RSA密钥默认的长度是512位，非常不安全。如果今天你的服务器上还是用512位的密钥，入侵者

可以先获取你的证书，使用暴力方式来算出对应的私钥，之后就可以冒充你的站点了。现在，一般认为2048位的RSA密钥是安全的，所以你应该采用这个长度的密钥。DSA密钥也应该不少于2048位，ECDSA密钥则应该是256位以上。

- 密码

  强烈建议使用密码去保存密钥，虽然这是只一个可选项。受密码保护的密钥可以被安全地存储、传输以及备份。但与此同时，这样的密钥也会带来不便，因为我们如果没有密码的话，就无法使用密钥了。例如每次你想要重启Web服务器的时候就会被要求输入密码。大多数情况下，这会导致极大的不便，在现实中几乎无法使用。如果真的发生入侵，在生产环境中使用密码保护过的密钥其实并没有提高安全性，因为一旦使用密码解密后，私钥会被明文保存在程序内存中，攻击者如果能登录服务器，那么只需要花费一点时间就可以很容易地获取到密钥。所以使用密码方式保护私钥只有在私钥并没有被放在生产环境服务器上的时候才有用。也就是说在生产环境上存放私钥和密码都是可以的。如果你想要让生产环境的私钥更加安全，那么需要考虑一下硬件解决方案。[①]

可以使用genrsa命令来生成RSA密钥：

```
$ openssl genrsa -aes128 -out fd.key 2048
Generating RSA private key, 2048 bit long modulus
....+++
.....................................................................................…
+++
e is 65537 (0x10001)
Enter pass phrase for fd.key: ****************
Verifying - Enter pass phrase for fd.key: ****************
```

这里，我指定私钥会使用AES-128算法来加密保存。当然也可以使用AES-192或者AES-256（分别使用开关-aes192和-aes256），但是最好不要使用其他算法（DES、3DES和SEED）。

> **警告**
>
> 上面输出结果中的e值表示公用指数，默认情况下会被设置为65 537。这是所谓的**短公用指数**（short public exponent），它可以显著提高RSA的验证性能。如果希望验证过程更快，请使用-3开关，选择3作为公用指数。但是历史上使用3作为公用指数有很多弱点，这就是所有人都建议你继续使用65 537的原因，后者被证明是安全和效率的一个平衡点。

私钥以所谓的PEM格式存储，该格式仅包含文本：

```
$ cat fd.key
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
```

① 一小部分组织会有非常严格的安全需求，要求不惜任何代价保证私钥安全。对他们来说解决方案是使用硬件安全模块（hardware security module，HSM），这类产品的设计使得即便能够物理访问它，也无法导出密钥。为实现此事，HSM不仅仅生成和存储密钥，还会执行所有必须的操作（例如，生成签名）。HSM非常昂贵。

```
DEK-Info: AES-128-CBC,01EC21976A463CE36E9DB59FF6AF689A

vERmFJzsLeAEDqWdXX4rNwogJp+y95uTnw+bOjWRw1+O1qgGqxQXPtH3LWDUz1Ym
mkpxmIwlSidVSUuUrrUzIL+V21EJ1W9iQ71SJoPOyzX7dYX5GCAwQm9Tsb40FhV/
[21 lines removed...]
4phGTprEnEwrffRnYrt7khQwrJhNsw6TTtthMhx/UCJdpQdaLW/TuylaJMWL1JRW
i321s5me5ej6Pr4fGccNOe7lZK+563d7v5znAx+Wo1C+F7YgF+g8LOQ8emC+6AVV
-----END RSA PRIVATE KEY-----
```

乍一看私钥是一堆随机数据，其实不是。你可以使用下面的rsa命令解析出私钥的结构：

```
$ openssl rsa -text -in fd.key
Enter pass phrase for fd.key: ****************
Private-Key: (2048 bit)
modulus:
    00:9e:57:1c:c1:0f:45:47:22:58:1c:cf:2c:14:db:
    [...]
publicExponent: 65537 (0x10001)
privateExponent:
    1a:12:ee:41:3c:6a:84:14:3b:be:42:bf:57:8f:dc:
    [...]
prime1:
    00:c9:7e:82:e4:74:69:20:ab:80:15:99:7d:5e:49:
    [...]
prime2:
    00:c9:2c:30:95:3e:cc:a4:07:88:33:32:a5:b1:d7:
    [...]
exponent1:
    68:f4:5e:07:d3:df:42:a6:32:84:8d:bb:f0:d6:36:
    [...]
exponent2:
    5e:b8:00:b3:f4:9a:93:cc:bc:13:27:10:9e:f8:7e:
    [...]
coefficient:
    34:28:cf:72:e5:3f:52:b2:dd:44:56:84:ac:19:00:
    [...]
writing RSA key
-----BEGIN RSA PRIVATE KEY-----
[...]
-----END RSA PRIVATE KEY-----
```

如果你需要单独查看密钥的公开部分，可以使用下面的rsa命令：

```
$ openssl rsa -in fd.key -pubout -out fd-public.key
Enter pass phrase for fd.key: ****************
```

如果你查看这个刚生成的文件，就会发现有明显的标识，表示这部分确实是公开的信息：

```
$ cat fd-public.key
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAnlccwQ9FRyJYHM8sFNsY
PUHJHJzhJdwcS7kBptutf/L6OvoEAzCVHi/m0qAA4QM5BziZgnvv+FNnE3sgE5pz
iovEHJ3C959mNQmpvnedXwfcOIlbrNqdISJiP0js6mDCzYjSO1NCQoy3UpYwvwj7
0ryR1F+abARehlts/Xs/PtX3VamrljiJN6JNgFICy3ZvEhLZEKxR7oob7TnyZDrj
IHxBbqPNzeiqLCFLFPGgJPa0cH8DdovBTesvu7wr/ecsf8CYyUCdEwGkZh9DKtdU
```

```
HFa9H8tWW2mX6uwYeHCnf2HTwOE8vjtOb8oYQxlQxtL7dpFyMgrpPOoOVkZZW/PO
NQIDAQAB
-----END PUBLIC KEY-----
```

一个好习惯是验证一下输出内容是否像你所期望的那样。例如，如果你忘记在命令行中包含-pubout开关，那么输出结果就会包括私钥信息而不是只有公钥。

DSA的密钥生成分成两个部分：先生成DSA的参数，然后再生成密钥。当然我倾向于使用一个命令来包括这两个步骤：

```
$ openssl dsaparam -genkey 2048 | openssl dsa -out dsa.key -aes128
Generating DSA parameters, 2048 bit long prime
This could take some time
[...]
read DSA key
writing DSA key
Enter PEM pass phrase: ****************
Verifying - Enter PEM pass phrase: ****************
```

这种方式可以让我生成一个受密码保护的密钥，而不会在磁盘上留下临时文件（DSA参数）或者临时的密钥。

创建ECDSA密钥的过程是类似的，但是不能创建任意长度的密钥。对于每个密钥，你需要选择一个命名曲线（named curve），它可以控制密钥长度，同时也限定了椭圆曲线的参数。下面的例子使用secp256r1这个命名曲线创建一个256位长度的ECDSA密钥：

```
$ openssl ecparam -genkey -name secp256r1 | openssl ec -out ec.key -aes128
using curve name prime256v1 instead of secp256r1
read EC key
writing EC key
Enter PEM pass phrase: ****************
Verifying - Enter PEM pass phrase: ****************
```

OpenSSL支持非常多的命名曲线（可以使用-list_curves开关获取完整的曲线列表），但是对于Web服务器使用的密钥来说，你只能使用两种：secp256r1（OpenSSL使用prime256v1作为名称）和secp384r1，因为只有这两个是大多数浏览器都支持的。

> **注意**
> 如果你使用OpenSSL 1.0.2，在生成密钥的时候可以使用genpkey命令节省大量的时间，因为它支持各种密钥类型以及配置参数。现在它是统一的密钥生成接口。

11.2.2　创建证书签名申请

一旦有了私钥，就可以创建证书签名申请（certificate signing request，CSR）。这是要求CA给证书签名的一种正式申请，该申请包含申请证书的实体的公钥以及该实体的某些信息。该数据将成为证书的一部分。CSR始终使用它携带的公钥所对应的私钥进行签名。

CSR创建的过程一般都是交互式的，你需要提供区分证书所需的不同元素。认真阅读openssl工具的帮助。如果你想让某一个字段为空，不要直接回车，必须输入一个点（.）；如果直接回车，

OpenSSL会直接使用这个字段默认的值（虽然几乎所有人都这么做，但是如果使用的是默认的OpenSSL配置，直接回车没有任何意义。只有在你意识到可以通过直接修改OpenSSL配置或者提供自己的配置文件来修改默认配置的时候，直接回车才是没问题的）。

```
$ openssl req -new -key fd.key -out fd.csr
Enter pass phrase for fd.key: ****************
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:GB
State or Province Name (full name) [Some-State]:.
Locality Name (eg, city) []:London
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Feisty Duck Ltd
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:www.feistyduck.com
Email Address []:webmaster@feistyduck.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

> **注意**
>
> 根据RFC 2985的5.4.1节[1]，**质询密码**（challenge password）是一个可选字段，用于在证书吊销过程中确认申请过该证书的最初实体的身份。如果输入这个字段，则会将密码包括在CSR文件中并发送给CA。几乎没有CA会依赖这个字段，我所看到的帮助信息都建议将这一字段留空，因为设置一个质询密码并没有增加CSR的任何安全性。另外不要将这个字段和密钥密码混淆了，它们的作用是不一样的。

CSR生成之后，可以使用它去直接进行证书签名或者将它发送给公共CA让他们对证书进行签名。下面会具体讲这两种方式，但是在操作之前，最好再检查一遍CSR是正确的。可以这么做：

```
$ openssl req -text -in fd.csr -noout
Certificate Request:
    Data:
        Version: 0 (0x0)
        Subject: C=GB, L=London, O=Feisty Duck Ltd, CN=www.feistyduck.com/emailAddress=webmaster@ ↵
feistyduck.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
```

---

[1] RFC 2985: PKCS #9: Selected Object Classes and Attribute Types Version 2.0，http://tools.ietf.org/html/rfc2985（M. Nystrom和B. Kaliski，2000年11月）。

```
                    00:b7:fc:ca:1c:a6:c8:56:bb:a3:26:d1:df:e4:e3:
                    [16 more lines...]
                    d1:57
                Exponent: 65537 (0x10001)
        Attributes:
            a0:00
    Signature Algorithm: sha1WithRSAEncryption
         a7:43:56:b2:cf:ed:c7:24:3e:36:0f:6b:88:e9:49:03:a6:91:
         [13 more lines...]
         47:8b:e3:28
```

11.2.3　用当前证书生成 CSR 文件

如果你想更新一张证书并且不想对里面的信息作任何更改，那么实现可以变得简单一些。使用下面的命令可以用当前的证书创建一个全新的CSR文件：

```
$ openssl x509 -x509toreq -in fd.crt -out fd.csr -signkey fd.key
```

> **注意**
> 除非是你在使用某种公钥钉扎的形式并且希望继续使用之前的密钥，否则建议每次申请新证书的时候都生成一个全新的密钥。密钥生成起来很快而且成本低廉，但是可以减少泄露的风险。

11.2.4　非交互方式生成 CSR

生成CSR并不一定要使用交互方式。使用自定义的OpenSSL配置文件，可以将这个过程自动化（本节中介绍的），并且还可做一些交互方式无法完成的事情（后续几节会介绍）。

例如我们想自动生成`www.feistyduck.com`的CSR文件，可以先创建一个fd.cnf文件：

```
[req]
prompt = no
distinguished_name = dn
req_extensions = ext
input_password = PASSPHRASE

[dn]
CN = www.feistyduck.com
emailAddress = webmaster@feistyduck.com
O = Feisty Duck Ltd
L = London
C = GB

[ext]
subjectAltName = DNS:www.feistyduck.com,DNS:feistyduck.com
```

然后使用下面的命令直接创建CSR文件：

```
$ openssl req -new -config fd.cnf -key fd.key -out fd.csr
```

11.2.5 自签名证书

如果你只是想安装一台自己使用的TLS服务器，那么可以不必找CA去获取一个公开信任的证书，自己就可以直接签发一个。最简单的方式就是生成自签名证书。如果你使用Firefox，那么可以在第一次访问网站的时候创建一个证书例外，然后就可以像使用公开可信证书一样正常访问了。

如果已经有了CSR，可以使用下面的文件创建证书：

```
$ openssl x509 -req -days 365 -in fd.csr -signkey fd.key -out fd.crt
Signature ok
subject=/CN=www.feistyduck.com/emailAddress=webmaster@feistyduck.com/O=Feisty Duck↩
Ltd/L=London/C=GB
Getting Private key
Enter pass phrase for fd.key: ****************
```

也可以无需单独创建一个CSR，使用下面的命令直接使用私钥创建自签名证书：

```
$ openssl req -new -x509 -days 365 -key fd.key -out fd.crt
```

如果你不想有交互提示，直接使用-subj并带上标题信息就可以了：

```
$ openssl req -new -x509 -days 365 -key fd.key -out fd.crt \
 -subj "/C=GB/L=London/O=Feisty Duck Ltd/CN=www.feistyduck.com"
```

11.2.6 创建对多个主机名有效的证书

默认情况下，OpenSSL创建的证书只包含一个公用名而且只能设置一个主机名。因为这个限制，即便你有其他相关联的站点，也不得不为每个站点生成一张单独的证书。在这种情况下，使用一张多域名（multidomain）的证书就有意义了。即便你是维护一个站点，也得确保用户在访问站点的所有子域名的时候证书是有效的。在实际使用中意味着使用至少两个名称，一个是使用www开头的，一个是没有任何前缀的（例如www.feistyduck.com和feistyduck.com）。

11

有两种方式在一张证书里面支持多主机名。一种方式是在X.509的使用者可选名称（subject alternative name，SAN）扩展字段里面列出所有要使用的主机名；另外一种就是使用泛域名。可以将两种方式合在一起，这样更加方便。在实际使用的时候，可以设置顶级域名和一个泛域名来囊括所有二级域名（例如feistyduck.com和*.feistyduck.com）。

> **警告**
> 当证书包括可选名称的时候，所有公用名就会被忽略。CA新创建的证书甚至可能不再包括任何公用名，所以，请在可选名称列表中包含所有想要的主机名。

首先，将扩展信息放在一个单独的文本文件中，我将该文件命名为fd.ext。在这个文件中，指定扩展的名称（subjectAltName），并且像下面这样列出需要的主机名：

```
subjectAltName = DNS:*.feistyduck.com, DNS:feistyduck.com
```

然后当使用x509命令签发证书的时候，使用-extfile开关引用该文件：

```
$ openssl x509 -req -days 365 \
-in fd.csr -signkey fd.key -out fd.crt \
-extfile fd.ext
```

剩下的步骤与之前的一样，当然在检查证书的时候你会发现它包括了SAN扩展信息：

```
X509v3 extensions:
         X509v3 Subject Alternative Name:
             DNS:*.feistyduck.com, DNS:feistyduck.com
```

11.2.7　检查证书

第一眼看到证书内容的时候会觉得它们就是一堆随机数据，而你只需要知道如何解析它们，就会发现其实里面包含了很多信息。x509命令可以帮助你查看刚生成的自签名证书。

下面的例子中我使用-text来打印证书内容，使用-noout则不打印编码后的证书内容，这样可以减少信息干扰（默认情况下会打印）：

```
$ openssl x509 -text -in fd.crt -noout
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 13073330765974645413 (0xb56dcd10f11aaaa5)
    Signature Algorithm: sha1WithRSAEncryption
        Issuer: CN=www.feistyduck.com/emailAddress=webmaster@feistyduck.com, ↩
O=Feisty Duck Ltd, L=London, C=GB
        Validity
            Not Before: Jun  4 17:57:34 2012 GMT
            Not After : Jun 4 17:57:34 2013 GMT
        Subject: CN=www.feistyduck.com/emailAddress=webmaster@feistyduck.com, ↩
O=Feisty Duck Ltd, L=London, C=GB
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:b7:fc:ca:1c:a6:c8:56:bb:a3:26:d1:df:e4:e3:
                    [16 more lines...]
                    d1:57
                Exponent: 65537 (0x10001)
    Signature Algorithm: sha1WithRSAEncryption
         49:70:70:41:6a:03:0f:88:1a:14:69:24:03:6a:49:10:83:20:
         [13 more lines...]
         74:a1:11:86
```

就像上面的例子一样，自签名证书一般只包括最基本的证书数据。相比而言，公共CA签发的证书则含有更多有意义的信息（通过X.509扩展机制），让我们快速了解一下。

基本约束（basic constraint）扩展用于标记证书是否是一个CA，这样的证书可以给其他证书进行签名。非CA证书则没有这个扩展项或者其中CA的值会被设置为FALSE。这是一个关键扩展，意味着所有软件必须识别这个字段。

```
X509v3 Basic Constraints: critical
```

```
    CA:FALSE
```

密钥用法（key usage，KU）和扩展密钥用法（extended key usage，EKU）扩展限制了证书的使用范围。如果这两个扩展存在，只有列表里面的使用方式是允许的。如果这个扩展不存在，则没有任何限制。你看到的这个例子是非常典型的Web服务器证书，也就是说它不能进行代码签名：

```
X509v3 Key Usage: critical
    Digital Signature, Key Encipherment
X509v3 Extended Key Usage:
    TLS Web Server Authentication, TLS Web Client Authentication
```

CRL分发点（CRL distribution point）扩展列出了CA证书吊销列表（certificate revocation list，CRL）的地址，当证书需要被吊销的时候这个信息非常重要。CA会对CRL进行签名，并且每隔一段时间发布一次（例如，7天）。

```
X509v3 CRL Distribution Points:
    Full Name:
      URI:http://crl.starfieldtech.com/sfs3-20.crl
```

> **注意**
>
> 你可能注意到了CRL的地址没有使用安全服务器，你可能会担心这样的话这个链接是否就不安全了。事实上不会的，因为每个CRL都由所签发的CA进行签名，浏览器可以验证CRL的完整性。事实上如果CRL由TLS进行分发，在验证CRL服务器所提供的证书是否吊销的时候，浏览器就会面临先有鸡还是先有蛋的问题。

证书策略（certificate policy）扩展用来指出证书使用哪种策略签发，例如扩展验证（extended validation，EV）标识就在这里（证书下面这个例子）。每个标识都拥有一个唯一的对象标识符（object identifier，OID），对签发的CA来说它们是唯一的。另外这个扩展一般还包括一个或者多个证书策略声明（certificate policy statement，CPS），一般都是网页或者PDF文档。

11

```
X509v3 Certificate Policies:
    Policy: 2.16.840.1.114414.1.7.23.3
    CPS: http://certificates.starfieldtech.com/repository/
```

颁发机构信息访问（authority information access，AIA）扩展包括了两个重要信息。首先它列出了CA的在线证书状态协议（online certificate status protocol，OCSP）响应程序的地址，可以用来实时监测证书的吊销情况。这个扩展可能还带上这个证书颁发者的证书地址（也就是证书链的上一层证书）。现在服务器证书已经很少直接使用根证书进行签名了，所以说用户一般需要在他们的配置里面加上一个或者多个中间证书，这时候就很容易出现因为漏了导致证书验证失败。有一些客户端（例如IE）会使用这个信息来获取中间CA的证书，从而弥补服务器忘记配置中间证书的问题，但是大部分的客户端不会。

```
Authority Information Access:
    OCSP - URI:http://ocsp.starfieldtech.com/
    CA Issuers - URI:http://certificates.starfieldtech.com/repository/sf_intermediate.crt
```

使用者密钥标识符（subject key identifier）和颁发机构密钥标识符（authority key identifier）扩展分别建立了唯一的使用者和颁发机构标识符。证书的颁发机构密钥标识符扩展的信息必须与颁发者的使用者密钥标识符扩展里面的信息一致。这些信息在证书链路径建立过程中相当有用，客户端会试图从分支（服务器）证书开始，寻找到根证书所有可能的路径。证书机构经常一个私钥对应多个证书，而这个字段允许软件可以非常可靠地让证书和密钥对应起来。现实中很多服务器提供的证书链其实都是错误的，但是因为浏览器可以自动寻找到其他可信的路径，所以这种情况常常被忽略了。

```
X509v3 Subject Key Identifier:
    4A:AB:1C:C3:D3:4E:F7:5B:2B:59:71:AA:20:63:D6:C9:40:FB:14:F1
X509v3 Authority Key Identifier:
    keyid:49:4B:52:27:D1:1B:BC:F2:A1:21:6A:62:7B:51:42:7A:8A:D7:D5:56
```

最后使用者可选名称（subject alternative name）扩展用来列出所有合法的主机名。这个扩展是可选的，如果不存在，客户端就会使用使用者（Subject）字段里面公用名（common name，CN）提供的信息；如果扩展存在，那么在验证过程中CN字段的内容会被忽略。

```
X509v3 Subject Alternative Name:
    DNS:www.feistyduck.com, DNS:feistyduck.com
```

11.2.8　密钥和证书格式转换

私钥和证书可以以各种格式进行存储，所以你可能经常需要进行各种格式之间的转换，最常见的格式如下所示。

- Binary (DER) certificate
  包含原始格式的X.509证书，使用DER ASN.1编码。
- ASCII (PEM) certificate(s)
  包含base64编码过的DER证书，它们以`-----BEGIN CERTIFICATE-----`开头，以`-----END CERTIFICATE-----`结尾。虽然有些程序可以允许多个证书存在一个文件中，但是一般来说一个文件只有一张证书。例如Apache Web服务器要求服务器的证书全部在一个文件里面，而中间证书一起放在另外一个文件中。
- Binary (DER) key
  包含DER ASN.1编码后的私钥的原始格式。OpenSSL使用他自己传统的方式创建密钥（SSLeay）格式。还有另外一种不常使用的格式叫作PKCS#8（RFC 5208定义的）。OpenSSL可以使用`pkcs8`命令进行PKCS#8格式的转换。
- ASCII (PEM) key
  包括base64编码后的DER密钥和一些元数据信息（例如密码的保存算法）。
- PKCS#7 certificate(s)
  RFC 2315定义的一种比较复杂的格式，设计的目的是用于签名和加密数据的传输。一般常见的是.p7b和.p7c扩展名的文件，并且文件里面可以包括所需的整个证书链。Java的密

钥管理工具支持这种格式。

- PKCS#12 (PFX) key and certificate(s)
  一种可以用来保存服务器私钥和整个证书链的复杂格式，一般以.p12和.pfx扩展名结尾。这类格式常见于Microsoft的产品，但是也用于客户端证书。虽然很久以前PFX表示PKCS#12之前的版本，现在PFX常被用作PKCS#12的代名词，不过你已经很难遇到老版本了。

1. PEM和DER转换

使用x509工具进行PEM和DER格式之间的证书转换，从PEM转换到DER：

```
$ openssl x509 -inform PEM -in fd.pem -outform DER -out fd.der
```

从DER转换到PEM：

```
$ openssl x509 -inform DER -in fd.der -outform PEM -out fd.pem
```

在私钥的DER和PEM格式之间进行转换方式是一样的，但是需要使用rsa或者dsa命令分别用作RSA和DSA密钥。

2. PKCS#12（PFX）转换

只需要一个命令就可以将PEM转换成PKCS#12。下面的例子将密钥（fd.key）、证书（fd.crt）以及中间证书（fd-chain.crt）转换成一个PKCS#12文件：

```
$ openssl pkcs12 -export \
    -name "My Certificate" \
    -out fd.p12 \
    -inkey fd.key \
    -in fd.crt \
    -certfile fd-chain.crt
Enter Export Password: ****************
Verifying - Enter Export Password: ****************
```


如果想反过来转换就没那么直接了，虽然也可以使用一个命令，但是这样结果也会存在一个文件里面：

```
$ openssl pkcs12 -in fd.p12 -out fd.pem -nodes
```

现在可以用你最喜欢的编辑器打开fd.pem然后手动将其分为独立的密钥、证书和中间证书文件，同时你会发现每一部分的前面都有额外的内容。例如：

```
Bag Attributes
    localKeyID: E3 11 E4 F1 2C ED 11 66 41 1B B8 83 35 D2 DD 07 FC DE 28 76
subject=/1.3.6.1.4.1.311.60.2.1.3=GB/2.5.4.15=Private Organization/serialNumber=06694169/C= ↩
   GB/ST=London/L=London/O=Feisty Duck Ltd/CN=www.feistyduck.com
issuer=/C=US/ST=Arizona/L=Scottsdale/O=Starfield Technologies, Inc./OU=http://certificates. ↩
starfieldtech.com/repository/CN=Starfield Secure CertificationAuthority
-----BEGIN CERTIFICATE-----
MIIF5zCCBM+gAwIBAgIHBG9JXlv9vTANBgkqhkiG9w0BAQUFADCB3DELMAkGA1UE
BhMCVVMxEDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTClNjb3R0c2RhbGUxJTAj
[...]
```

多出来的这些元数据可以非常方便地用来识别证书。显然你需要确保主证书文件囊括了分支服务器证书而不是别的证书，并且你还得确保中间证书放置的顺序是正确的，签发证书应该在被签名证书的后面。如果你看到了自签名的根证书，可以直接删除或者将它存到别的地方，不应该将其放到证书链里面。

> **警告**
> 除了编码后的密钥和证书，最终转换后的结果不应该包括任何别的内容。虽然有一些工具可以自动忽略那些不需要的部分，但不是所有工具都可以。如果在PEM保留了这些无用的内容，可能会导致后续排查问题的时候遇到困难。

可以直接让OpenSSL来划分不同的部分，不过需要执行多次的pkcs12命令（每次还得输入对应的密码）：

```
$ openssl pkcs12 -in fd.p12 -nocerts -out fd.key -nodes
$ openssl pkcs12 -in fd.p12 -nokeys -clcerts -out fd.crt
$ openssl pkcs12 -in fd.p12 -nokeys -cacerts -out fd-chain.crt
```

这种方式并没有更简单，你还是得检查每个文件确保它们的内容是正确的，并且已经移除了那些元数据。

3. PKCS#7转换

使用crl2pkcs7命令将PEM转换成PKCS#7格式：

```
$ openssl crl2pkcs7 -nocrl -out fd.p7b -certfile fd.crt -certfile fd-chain.crt
```

将pkcs7命令与-print_certs开关一起使用可以将PKCS#7转换成PEM格式：

```
openssl pkcs7 -in fd.p7b -print_certs -out fd.pem
```

与PKCS#12一样，你还得手动编辑fd.pem文件并且将其分成不同部分。

11.3　配置

本节我会讨论两个与TLS部署相关的话题。首先是密码套件的配置，你需要从可用的TLS密码套件里面找出你希望用来加密通信的那些套件。这个话题很重要，因为几乎每个使用OpenSSL的程序都会重用它的套件配置，这意味着你只需要学会如何给某一个程序配置密码套件后，别的地方同时也可以适用。第二个话题主要讨论纯加解密操作的性能衡量方式。

11.3.1　选择密码套件

TLS服务器配置中一个共同的任务是选择支持哪些密码套件。依赖OpenSSL的那些程序一般采用与OpenSSL同样的套件配置方式，只需要简单地将配置传递给OpenSSL就可以了。例如在Apache httpd中，它的密码套件配置可能类似于这样：

```
SSLHonorCipherOrder On
SSLCipherSuite "HIGH:!aNULL:@STRENGTH"
```

第一行控制密码套件的优先级（同时让httpd主动选择套件）。

第二行控制支持的套件列表。

因为有很多细节要考虑，所以要想配置合理的套件需要花费不少的时间。最好的方式是使用OpenSSL的ciphers命令来确定每个套件对应的配置。

1. 获取所支持的套件列表

在开始配置之前，需要先确定你安装的OpenSSL版本支持哪些套件。可以调用带有-v开关和ALL:COMPLEMENTOFALL参数的ciphers命令来列出所有套件（显然这里ALL并不是真正意义上的“所有”）：

```
$ openssl ciphers -v 'ALL:COMPLEMENTOFALL'
ECDHE-RSA-AES256-GCM-SHA384   TLSv1.2  Kx=ECDH   Au=RSA    Enc=AESGCM(256)  Mac=AEAD
ECDHE-ECDSA-AES256-GCM-SHA384 TLSv1.2  Kx=ECDH   Au=ECDSA  Enc=AESGCM(256)  Mac=AEAD
ECDHE-RSA-AES256-SHA384       TLSv1.2  Kx=ECDH   Au=RSA    Enc=AES(256)     Mac=SHA384
ECDHE-ECDSA-AES256-SHA384     TLSv1.2  Kx=ECDH   Au=ECDSA  Enc=AES(256)     Mac=SHA384
ECDHE-RSA-AES256-SHA          SSLv3    Kx=ECDH   Au=RSA    Enc=AES(256)     Mac=SHA1
[106 more lines...]
```

> **提示**
>
> 如果使用的是OpenSSL 1.0.0之后的版本，可以使用大写的-V开关来获取更详细的输出内容。这种模式下的输出内容会包括套件的ID，非常方便。例如OpenSSL不一定使用RFC标准的套件名称，在这种情况下就需要通过ID进行二次确认。

我这里一共输出了111个套件，每一行都包含一个套件的信息和下面这些信息：

(1) 套件名称

(2) 最低的TLS版本

(3) 密钥交换算法

(4) 密钥验证算法

(5) 对称加密算法和长度

(6) 消息摘要（完整性检查）算法

(7) 出口套件指示符

如果不使用ALL:COMPLEMENTOFALL而是使用别的密码套件参数，OpenSSL就只会列出对应的套件。例如你可以只显示基于RC4的那些密码套件，如下所示。

```
$ openssl ciphers -v 'RC4'
ECDHE-RSA-RC4-SHA    SSLv3  Kx=ECDH        Au=RSA    Enc=RC4(128)  Mac=SHA1
ECDHE-ECDSA-RC4-SHA  SSLv3  Kx=ECDH        Au=ECDSA  Enc=RC4(128)  Mac=SHA1
AECDH-RC4-SHA        SSLv3  Kx=ECDH        Au=None   Enc=RC4(128)  Mac=SHA1
ADH-RC4-MD5          SSLv3  Kx=DH          Au=None   Enc=RC4(128)  Mac=MD5
ECDH-RSA-RC4-SHA     SSLv3  Kx=ECDH/RSA    Au=ECDH   Enc=RC4(128)  Mac=SHA1
ECDH-ECDSA-RC4-SHA   SSLv3  Kx=ECDH/ECDSA  Au=ECDH   Enc=RC4(128)  Mac=SHA1
RC4-SHA              SSLv3  Kx=RSA         Au=RSA    Enc=RC4(128)  Mac=SHA1
RC4-MD5              SSLv3  Kx=RSA         Au=RSA    Enc=RC4(128)  Mac=MD5
PSK-RC4-SHA          SSLv3  Kx=PSK         Au=PSK    Enc=RC4(128)  Mac=SHA1
EXP-ADH-RC4-MD5      SSLv3  Kx=DH(512)     Au=None   Enc=RC4(40)   Mac=MD5  export
```

```
EXP-RC4-MD5          SSLv3     Kx=RSA(512)    Au=RSA      Enc=RC4(40)     Mac=MD5    export
```

虽然有一些套件是不安全的，OpenSSL还是会把你要求的套件全部输出来，当然你还得从这些配置里面筛选出安全的套件。而且套件的输出顺序也是有含义的，当你的TLS服务器设置为手动选择密码套件时（这是最好的方式，也应该这么设置），越靠前的套件优先级越高。

2. 关键字

密码套件关键字（keyword）是密码套件配置的最基本的组成部分，每个套件名称（例如，RC4-SHA）对应一个套件。其他关键字则根据某些标准对应一组套件。关键字名称是区分大小写的。我本来应该直接向你呈现记录有一堆复杂关键字的OpenSSL文档，但是发现加密文档不是最新的，缺少了最近新增的内容。所以我会在本节尝试记录下所有关键字。

组关键字是选择常用密码套件的一种快捷方式，例如HIGH只会列出安全性非常强的密码套件，如表11-1所示。

表11-1 组关键字

关 键 字	含 义
DEFAULT	默认的密码列表。这是在编译的时候确定的，OpenSSL 1.0.0的默认值一般是ALL:!aNULL:!eNULL。这一定是第一个特定的密码字符串
COMPLEMENTOFDEFAULT	这些密码包含在ALL里面，但是默认是未启用的。当前是ADH。需要注意的是这条规则不包含eNULL，也同时没有被ALL所囊括在内（如果有必要，请使用COMPLEMENTOFALL）
ALL	除了eNULL密码之外的所有密码套件，需要显式启用才可以使用
COMPLEMENTOFALL	不包含在ALL的密码套件中，当前是eNULL
HIGH	“High”加密密码套件。当前是指密钥长度超过128位的密码套件
MEDIUM	“Medium”加密密码套件，当前是指那些使用128位的加密算法
LOW	“Low”加密密码套件，当前是指使用64或者56位的加密算法，但是不包含那些出口的密码套件。不安全
EXP、EXPORT	出口加密算法。包含40和56位的算法。不安全
EXPORT40	40位出口加密算法。不安全
EXPORT56	56位出口加密算法。不安全
TLSv1、SSLv3、SSLv2	分别是TLS 1.0、SSL 3或者SSL 2支持的密码套件

摘要关键字会按照特定的摘要算法筛选出密码套件。例如MD5表示筛选出所有基于MD5作为完整性验证的密码套件，如表11-2所示。

表11-2 摘要算法关键字

关 键 字	含 义
MD5	使用MD5的密码套件。已废弃而且不安全
SHA、SHA1	使用SHA1的密码套件
SHA256(v1.0.0+)	使用SHA256的密码套件
SHA384(v1.0.0+)	使用SHA384的密码套件

注意

摘要算法的关键字只筛选出在协议层会进行数据完整性验证的套件。TLS 1.2引入了对可验证加密的支持，它是一种将加密和完整性检查打包在一起的机制。当使用了AEAD（authenticated encryption with associated data）之后，TLS协议就不需要提供额外的完整性检查。因此你无法通过摘要算法筛选出AEAD套件（这些套件现在的名称里面会带上GCM）。这些套件名称虽然以SHA256和SHA384结尾，但是这里它们仅仅用来生成套件所需要的**伪随机函数**（pseudorandom function）。

验证关键字会筛选出那些基于它们所用验证方式的套件（参见表11-3）。现在几乎所有的公用证书都使用RSA作为验证。未来我们也许会看到越来越多的人使用椭圆曲线（elliptic curve，ECDSA）证书。

表11-3　验证关键字

关　键　字	含　　义
aDH	使用DH验证的密码套件，即证书携带了DH密钥（v1.0.2+）
aDSS、DSS	使用DSS验证的密码套件，即证书携带了DSS密钥
aECDH (v1.0.0+)	使用ECDH验证的密码套件
aECDSA (v1.0.0+)	使用ECDSA验证的密码套件
aNULL	不支持验证的密码套件。现在是指匿名DH算法。不安全
aRSA	使用RSA验证的密码套件，即证书携带了RSA密钥
PSK	使用PSK（Pre-Shared密钥）进行验证的密码套件
SRP	使用SRP（安全远程密码）进行验证的密码套件

密钥交换关键字会基于所选择的密钥交换算法筛选套件（参见表11-4）。当使用临时Diffie-Hellman套件的时候，OpenSSL对套件和关键字的命名总是不一致。在套件名称里面，临时套件会将E放在密钥交换算法的后面（例如，ECDHE-RSA-RC4-SHA和DHE-RSA-AES256-SHA），但是关键字则是放在开头（例如，EEDHE和EDH）。更糟糕的是，有些老的套件会将E放在密钥交换算法的开头（例如，EDH-RSA-DES-CBC-SHA）。

表11-4　密钥交换关键字

关　键　字	含　　义
ADH	匿名DH密码套件。不安全
AECDH(v1.0.0+)	匿名ECDH密码套件。不安全
DH	使用DH的密码套件（包括临时和匿名DH）
ECDH(v1.0.0+)	使用ECDH的密码套件（包括临时和匿名ECDH）
EDH(v1.0.0+)	使用临时DH进行密码协商的密码套件
EECDH (v1.0.0+)	使用临时ECDH的密码套件
kECDH (v1.0.0+)	使用ECDH密码协商的密码套件
kEDH	使用临时DH密钥协商的密码套件（包括匿名DH）

11

（续）

关 键 字	含 义
kEECDH (v1.0.0+)	使用临时ECDH密钥协商的密码套件（包括匿名ECDH）
kRSA、RSA	使用RSA进行密钥交换的密码套件

密码关键字基于它们使用的密码选择套件（参见表11-5）。

表11-5 密码关键字

关 键 字	含 义
3DES	使用三倍DES的密码套件
AES	使用AES的密码套件
AESGCM (v1.0.0+)	使用AES GCM的密码套件
CAMELLIA	使用Camellia的密码套件
DES	使用单独DES的密码套件。已废弃而且不安全
eNULL、NULL	不加密的密码套件。不安全
IDEA	使用IDEA的密码套件
RC2	使用RC2的密码套件。已废弃而且不安全
RC4	使用RC4的密码套件。不安全
SEED	使用SEED的密码套件

还有一些套件无法归到任何类别里面（参见表11-6）。它们中的大部分与GOST标准相关，是在前苏联解体之后，独联体中的一些国家开发出来的。

表11-6 其他关键字

关 键 字	含 义
@STRENGTH	根据加密算法密钥长度进行排序后的密码套件列表
aGOST	使用GOST R 34.10（要么2001，要么94）进行验证的密码套件，需要有GOST功能的引擎
aGOST01	使用GOST R34.10-2001验证的密码套件
aGOST94	使用GOST R 34.10-94验证的密码套件。已废弃。请使用GOST R 34.10-2001
kGOST	使用VKO 34.10进行密钥交换的密码套件，已在RFC 4357中指定
GOST94	基于GOST R 34.11-94使用HMAC的密码套件
GOST89MAC	使用GOST 28147-89而且不是HMAC的密码套件

3. 组合关键字

大部分情况下使用直接关键字本身即可，但是有时候也需要使用+号将两个或者多个关键字组合在一起来筛选出符合多个要求的套件。下面的例子中我们筛选出RC4和SHA的套件：

```
$ openssl ciphers -v 'RC4+SHA'
ECDHE-RSA-RC4-SHA     SSLv3 Kx=ECDH      Au=RSA    Enc=RC4(128)  Mac=SHA1
ECDHE-ECDSA-RC4-SHA   SSLv3 Kx=ECDH      Au=ECDSA  Enc=RC4(128)  Mac=SHA1
AECDH-RC4-SHA         SSLv3 Kx=ECDH      Au=None   Enc=RC4(128)  Mac=SHA1
ECDH-RSA-RC4-SHA      SSLv3 Kx=ECDH/RSA  Au=ECDH   Enc=RC4(128)  Mac=SHA1
```

```
ECDH-ECDSA-RC4-SHA    SSLv3 Kx=ECDH/ECDSA  Au=ECDH     Enc=RC4(128)   Mac=SHA1
RC4-SHA               SSLv3 Kx=RSA         Au=RSA      Enc=RC4(128)   Mac=SHA1
PSK-RC4-SHA           SSLv3 Kx=PSK         Au=PSK      Enc=RC4(128)   Mac=SHA1
```

4. 创建密码套件列表

构建一份密码套件配置的关键在于当前套件列表（current suite list）。这份列表最开始都是空的，没有任何套件，不过当你给配置字符串添加任何关键字之后这份列表就会发生变化。默认情况下，新套件会追加到结尾处。例如要选择RC4和AES算法的套件可以是：

```
$ openssl ciphers -v 'RC4:AES'
```

冒号常用来分隔关键字，空格和逗号也有同样的作用。下面的命令与之前命令的效果一样：

```
$ openssl ciphers -v 'RC4 AES'
```

5. 关键字修饰符

关键字修饰符是那些可以放在每个关键字的前面的字符，它可以改变默认行为（默认行为是加入到列表里面），支持下面这些操作。

- 结尾附加

  在列表末尾加入套件。列表上已经有的套件会保持原来的位置。当关键字之前没有别的修饰符时，这是默认行为。
- 删除（-）

  从列表中删除所有匹配的套件，但是后续其他关键字依旧可以让这些套件重现。
- 永久删除（!）

  从列表中删除所有匹配的套件并且后续任何关键字也无法重现这些套件。当你确定永远不再使用某些套件的情况下，使用这个修饰符可以让你后续的决策变得简单，同时可以减少犯错。
- 移到行尾（+）

  将所有符合的套件移到列表末尾。仅对已经存在的套件有效，不会新增套件到列表里面。在你希望保留某些弱套件同时希望优先使用强密码套件的时候，就可以使用这种方式。例如RC4:+MD5会保留所有的RC4套件，但是将其中基于MD5的套件移到末尾。

- 排序

@STRENGTH和别的关键字都不一样（我猜测这也是为什么它的名称里面有一个@）：它不会增加或者减少任何套件，但是会按照套件的加密强度进行降序排列。自动排序是一个很好的想法，但是在现实中我们很难通过加密长度就判断出密码套件的强度。

例如下面的配置：

```
$ openssl ciphers -v 'DES-CBC-SHA:DES-CBC3-SHA:RC4-SHA:AES256-SHA:@STRENGTH'
AES256-SHA                  SSLv3   Kx=RSA  Au=RSA   Enc=AES(256)    Mac=SHA1
DES-CBC3-SHA                SSLv3   Kx=RSA  Au=RSA   Enc=3DES(168)   Mac=SHA1
RC4-SHA                     SSLv3   Kx=RSA  Au=RSA   Enc=RC4(128)    Mac=SHA1
DES-CBC-SHA                 SSLv3   Kx=RSA  Au=RSA   Enc=DES(56)     Mac=SHA1
```

理论上，输出是按照强度排序的，现实中你会希望能够更好地控制套件顺序：

- 例如当使用TLS 1.0以及更低版本的时候，AES256-SHA（CBC套件）会遭受到BEAST攻击，这时候你会希望提高RC4-SHA套件的优先级，它不受BEAST攻击的影响。
- 3DES表面上是168位强度，实际上一种叫作中途相遇（meet-in-the-middle）的攻击可以将它的强度降到112位，[①]加上其他的问题甚至可以将强度降为108位。[②]实际上DES-CBC3-SHA应该被归为128位密码套件，严格来说OpenSSL将3DES归为168位密码套件其实是一个bug，未来有可能会修复。

6. 处理错误

你在配置的时候可能会遇到两种错误，第一种是排版错误或者使用了不存在的关键字：

```
$ openssl ciphers -v '@HIGH'
Error in cipher list
140460843755168:error:140E6118:SSL routines:SSL_CIPHER_PROCESS_RULESTR:invalid command:ssl_ciph.
    c:1317:
```

这个输出结果非常含糊，但是的确包含了错误信息。

另外一种错误是出现类似下面的结果，一般是因为不存在匹配的密码套件。

```
$ openssl ciphers -v 'SHA512'
Error in cipher list
140202299557536:error:1410D0B9:SSL routines:SSL_CTX_set_cipher_list:no cipher match:ssl_lib.c:1312:
```

7. 合并多个套件

为了说明如何将多个套件的功能组合在一起，我会举一个现实的例子；但是请记住下面仅仅是一个例子。因为在进行配置之后我们有很多东西需要考虑，所以不存在一个配置可以匹配所有情况。

因此在开始配置之前，我们需要清晰地确定想要达成什么效果。对我来说，我希望有下面这样一个相对安全同时高性能的配置。

(1) 只使用加密效果128位及以上的密码套件（这就排除掉了3DES）。

(2) 只使用提供强验证的套件（这就排除掉了匿名和出口套件）。

(3) 不要使用任何依赖不安全算法的套件。

(4) 实现无论是用什么密钥和协议，都可以支持前向保密（forward secrecy）。这个要求会让我们无法使用RSA进行高性能的密钥交换，因此会导致一部分的性能损失。通过将更快的ECDHE的优先级提到DHE之前，可以尽可能地降低损耗。

(5) 优先使用ECDSA而不是RSA。这项要求需要我们支持双密钥部署模式，因为这样我们才能尽可能地使用ECDSA算法，同时又可以支持那些只能使用RSA的客户端。

(6) 支持TLS 1.2的客户端会优先使用AES GCM套件，这个套件是在当前TLS版本中安全性最

① Cryptography/Meet In The Middle Attack，https://en.wikibooks.org/wiki/Cryptography/Meet_In_The_Middle_Attack（Wikibooks，检索于2014年3月31日）。

② Attacking Triple Encryption，http://th.informatik.uni-mannheim.de/people/lucks/publ.shtml（Stefan Lucks，1998）。

好的。

(7) 最近RC4有更多弱点被发现[1]，所以我将它放到了列表的末尾，当然最好是去掉它。虽然BEAST在某些情况下依旧是一个问题，我假设这个问题已经由客户端解决掉了。

从一开始就应该禁用掉所有不希望使用的套件，这样就可以减少很多配置，而且避免了不小心错误地引入不该使用的套件。

使用下面这些加密字符串可以筛选出弱密码套件。

- ❑ `aNULL` 无验证
- ❑ `eNULL` 无加密
- ❑ `LOW` 低强度套件
- ❑ `3DES` 有效加密强度为108位
- ❑ `MD5` 使用MD5的套件
- ❑ `EXP` 已废弃的出口套件

因为DSA、PSK、SRP和ECDH很少使用，所以我直接将它们去除，这样也可以减少出现的套件的数量。OpenSSL虽然还支持IDEA和SEED，但它们都已经是废弃的算法了。在我的配置里也去掉了CAMELLIA，因为它比较慢而且不如AES支持的客户端多（例如在实际中不支持GCM或者ECDHE变体）。

```
!aNULL !eNULL !LOW !3DES !MD5 !EXP !DSS !PSK !SRP !kECDH !CAMELLIA !IDEA !SEED
```

现在我们将注意力聚焦在我们想要的效果上。因为前向保密是我们的最高优先级，所以我们以kEECHE和kEDH关键字开头：

```
kEECDH kEDH !aNULL !eNULL !LOW !3DES !MD5 !EXP !DSS !PSK !SRP !kECDH !CAMELLIA
!IDEA !SEED
```

你会发现如果直接使用上面的配置，RSA套件会排在前面，但其实我希望将ECDSA放在最前面：

```
ECDHE-RSA-AES256-GCM-SHA384      TLSv1.2   Kx=ECDH    Au=RSA     Enc=AESGCM(256)  Mac=AEAD
ECDHE-ECDSA-AES256-GCM-SHA384    TLSv1.2   Kx=ECDH    Au=ECDSA   Enc=AESGCM(256)  Mac=AEAD
ECDHE-RSA-AES256-SHA384          TLSv1.2   Kx=ECDH    Au=RSA     Enc=AES(256)     Mac=SHA384
ECDHE-ECDSA-AES256-SHA384        TLSv1.2   Kx=ECDH    Au=ECDSA   Enc=AES(256)     Mac=SHA384
ECDHE-RSA-AES256-SHA             SSLv3     Kx=ECDH    Au=RSA     Enc=AES(256)     Mac=SHA1
ECDHE-ECDSA-AES256-SHA           SSLv3     Kx=ECDH    Au=ECDSA   Enc=AES(256)     Mac=SHA1
ECDHE-RSA-AES128-GCM-SHA256      TLSv1.2   Kx=ECDH    Au=RSA     Enc=AESGCM(128)  Mac=AEAD
[...]
```

因此我可以将kEECDH+ECDSA放在配置的最前面，这样就可以将ECDSA优先筛选出来了：

```
kEECDH+ECDSA kEECDH kEDH !aNULL !eNULL !LOW !3DES !MD5 !EXP !DSS !PSK !SRP !kECDH!CAMELLIA↩
!IDEA !SEED
```

下面一个问题是老的套件（SSL 3）和新的套件混在一起了（TLS 1.2）。为了尽可能安全，我希望所有的TLS 1.2的客户端都能协商并使用TLS 1.2的密码套件。因此可以通过使用+SHA关键字

[1] On the Security of RC4 in TLS and WPA，http://www.isg.rhul.ac.uk/tls/（AlFardan等，2013年3月13日）。

来将老的套件放到列表的末尾（TLS 1.2套件只会使用SHA256和SHA384，所以它们不会匹配到）。

```
kEECDH+ECDSA kEECDH kEDH +SHA !aNULL !eNULL !LOW !3DES !MD5 !EXP !DSS !PSK !SRP !kECDH !CAMELLIA↩
!IDEA !SEED
```

到这里基本完成，只需要在使用HIGH关键字将剩余的安全套件也加入到列表末尾即可。另外，还需要确保RC4套件在列表的末尾，可以使用+RC4（将RC4套件放到列表末尾）和RC4（将剩余的未在列表里面的RC4套件加入进来）：

```
kEECDH+ECDSA kEECDH kEDH HIGH +SHA +RC4 RC4 !aNULL !eNULL !LOW !3DES !MD5 !EXP!DSS !PSK !SRP↩
!kECDH !CAMELLIA !IDEA !SEED
```

我们再看一下最后的输出结果，一共由28个套件组成。第一组都是TLS 1.2的套件：

```
ECDHE-ECDSA-AES256-GCM-SHA384   TLSv1.2  Kx=ECDH   Au=ECDSA  Enc=AESGCM(256) Mac=AEAD
ECDHE-ECDSA-AES256-SHA384       TLSv1.2  Kx=ECDH   Au=ECDSA  Enc=AES(256)    Mac=SHA384
ECDHE-ECDSA-AES128-GCM-SHA256   TLSv1.2  Kx=ECDH   Au=ECDSA  Enc=AESGCM(128) Mac=AEAD
ECDHE-ECDSA-AES128-SHA256       TLSv1.2  Kx=ECDH   Au=ECDSA  Enc=AES(128)    Mac=SHA256
ECDHE-RSA-AES256-GCM-SHA384     TLSv1.2  Kx=ECDH   Au=RSA    Enc=AESGCM(256) Mac=AEAD
ECDHE-RSA-AES256-SHA384         TLSv1.2  Kx=ECDH   Au=RSA    Enc=AES(256)    Mac=SHA384
ECDHE-RSA-AES128-GCM-SHA256     TLSv1.2  Kx=ECDH   Au=RSA    Enc=AESGCM(128) Mac=AEAD
ECDHE-RSA-AES128-SHA256         TLSv1.2  Kx=ECDH   Au=RSA    Enc=AES(128)    Mac=SHA256
DHE-RSA-AES256-GCM-SHA384       TLSv1.2  Kx=DH     Au=RSA    Enc=AESGCM(256) Mac=AEAD
DHE-RSA-AES256-SHA256           TLSv1.2  Kx=DH     Au=RSA    Enc=AES(256)    Mac=SHA256
DHE-RSA-AES128-GCM-SHA256       TLSv1.2  Kx=DH     Au=RSA    Enc=AESGCM(128) Mac=AEAD
DHE-RSA-AES128-SHA256           TLSv1.2  Kx=DH     Au=RSA    Enc=AES(128)    Mac=SHA256
AES256-GCM-SHA384               TLSv1.2  Kx=RSA    Au=RSA    Enc=AESGCM(256) Mac=AEAD
AES256-SHA256                   TLSv1.2  Kx=RSA    Au=RSA    Enc=AES(256)    Mac=SHA256
AES128-GCM-SHA256               TLSv1.2  Kx=RSA    Au=RSA    Enc=AESGCM(128) Mac=AEAD
AES128-SHA256                   TLSv1.2  Kx=RSA    Au=RSA    Enc=AES(128)    Mac=SHA256
```

优先使用ECDHE，然后是DHE，最后才是剩余的TLS 1.2套件。在任何一组里面，ECDSA和GCM都拥有更高的优先级。

第二组是TLS 1.0客户端会使用的套件，使用类似第一组的优先级顺序：

```
ECDHE-ECDSA-AES256-SHA          SSLv3    Kx=ECDH   Au=ECDSA  Enc=AES(256)    Mac=SHA1
ECDHE-ECDSA-AES128-SHA          SSLv3    Kx=ECDH   Au=ECDSA  Enc=AES(128)    Mac=SHA1
ECDHE-RSA-AES256-SHA            SSLv3    Kx=ECDH   Au=RSA    Enc=AES(256)    Mac=SHA1
ECDHE-RSA-AES128-SHA            SSLv3    Kx=ECDH   Au=RSA    Enc=AES(128)    Mac=SHA1
DHE-RSA-AES256-SHA              SSLv3    Kx=DH     Au=RSA    Enc=AES(256)    Mac=SHA1
DHE-RSA-AES128-SHA              SSLv3    Kx=DH     Au=RSA    Enc=AES(128)    Mac=SHA1
DHE-RSA-SEED-SHA                SSLv3    Kx=DH     Au=RSA    Enc=SEED(128)   Mac=SHA1
AES256-SHA                      SSLv3    Kx=RSA    Au=RSA    Enc=AES(256)    Mac=SHA1
AES128-SHA                      SSLv3    Kx=RSA    Au=RSA    Enc=AES(128)    Mac=SHA1
```

最后把RC4套件放到末尾：

```
ECDHE-ECDSA-RC4-SHA             SSLv3    Kx=ECDH   Au=ECDSA  Enc=RC4(128)    Mac=SHA1
ECDHE-RSA-RC4-SHA               SSLv3    Kx=ECDH   Au=RSA    Enc=RC4(128)    Mac=SHA1
RC4-SHA                         SSLv3    Kx=RSA    Au=RSA    Enc=RC4(128)    Mac=SHA1
```

8. 推荐配置

前面一节展示了一个使用OpenSSL套件关键字生成密码套件配置的例子，但这不是最好的方

式。事实上没有任何一个配置可以满足所有人。在本节中我会基于你的偏好和风险评定，提供几个不同的配置供你选择。

本节设计的配置原理与前面的方式一样，但是我会修改两个地方来获得更好的性能。首先我会将128位的套件放在列表的前面，因为虽然256位的套件安全性更高，但是128位在当前已经足够安全，所以使用256位的意义不大，而且还会造成性能损失。另外我倾向于选择HMAC-SHA而不是HMAC-SHA256或者HMAC-SHA384,后面两个更慢而且所提高的安全性当前看来意义不大，HMAC-SHA就已经足够了。

另外我将由原来用关键字进行配置的方式改成直接使用套件名称。使用关键字不是个坏主意，你只需要确定安全的需求，剩下的由OpenSSL库去帮你选择，你甚至不需要知道会使用哪些套件。在实际中这种方式并不理想，因为我们对所需要的套件要求越来越严格，特别是我们有时候就想使用其中某个套件，以及按照我们所希望的顺序排列。

在配置里面直接使用套件名称还有一个好处：你只需要列出你想要的套件即可；而且如果你看了别人的配置，就可以直接通过套件名称确定它们使用的套件，不再需要通过OpenSSL执行关键字命令后才能获取。

下面是我默认开始的配置，同时满足了强加密和高性能。

```
ECDHE-ECDSA-AES128-GCM-SHA256
ECDHE-ECDSA-AES256-GCM-SHA384
ECDHE-ECDSA-AES128-SHA
ECDHE-ECDSA-AES256-SHA
ECDHE-ECDSA-AES128-SHA256
ECDHE-ECDSA-AES256-SHA384
ECDHE-RSA-AES128-GCM-SHA256
ECDHE-RSA-AES256-GCM-SHA384
ECDHE-RSA-AES128-SHA
ECDHE-RSA-AES256-SHA
ECDHE-RSA-AES128-SHA256
ECDHE-RSA-AES256-SHA384
DHE-RSA-AES128-GCM-SHA256
DHE-RSA-AES256-GCM-SHA384
DHE-RSA-AES128-SHA
DHE-RSA-AES256-SHA
DHE-RSA-AES128-SHA256
DHE-RSA-AES256-SHA256
EDH-RSA-DES-CBC3-SHA
```

上面这份配置只提供了支持前向保密和强加密的套件。当前流行的浏览器和客户端都能支持，但是一些老的客户端，例如运行在Windows XP上的那些老版本IE就会失败。如果真的需要支持那些非常老的客户端，那只能考虑在末尾加上下面的套件：

```
AES128-SHA
AES256-SHA
DES-CBC3-SHA
ECDHE-RSA-RC4-SHA
RC4-SHA
```

这些老旧套件大部分使用RSA作为密钥交换，因为无法提供前向保密的功能。我们将AES放在前面，但是为了最大限度地支持更多的客户端，我们也将3DES和不安全的RC4加了进去。如果无法避免使用RC4，最好用ECDHE套件来提供前向保密。

11.3.2　性能

你可能知道，计算速度是任何加密操作最大的影响因子。OpenSSL自带了性能评测工具，你可以使用它对系统的能力和上限进行一个大致的评定。你可以使用speed命令来执行评测。

如果调用speed命令的时候没有带上任何参数，OpenSSL会输出很多无用信息，最好的方式是只测试你关心的那些套件。从Web服务器安全考虑，你可能会关心RC4、AES、RSA、ECDH和SHA算法。

```
$ openssl speed rc4 aes rsa ecdh sha
```

输出一共包括三个部分，第一部分包括OpenSSL的版本信息和编译时间。如果你在测试不同版本的OpenSSL，并且编译时的选项也不一样，那么这些信息将会很有帮助：

```
OpenSSL 0.9.8k 25 Mar 2009
built on: Wed May 23 00:02:00 UTC 2012
options:bn(64,64) md2(int) rc4(ptr,char) des(idx,cisc,16,int) aes(partial) blowfish(ptr2)
compiler: cc -fPIC -DOPENSSL_PIC -DZLIB -DOPENSSL_THREADS -D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H↩
-m64-DL_ENDIAN -DTERMIO -O3 -Wa,--noexecstack -g -Wall -DMD32_REG_T=int -DOPENSSL_BN_ASM_MONT↩
-DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DMD5_ASM -DAES_ASM
available timing options: TIMES TIMEB HZ=100 [sysconf value]
timing function used: times
The 'numbers' are in 1000s of bytes per second processed.
```

第二部分包含对称密码性能数据（例如，散列函数和私有密码算法）：

```
type            16 bytes       64 bytes       256 bytes    1024 bytes     8192 bytes
sha1            29275.44k      85281.86k      192290.28k    280526.68k     327553.12k
rc4             160087.81k     172435.03k     174264.75k    176521.50k     176700.62k
aes-128 cbc     90345.06k      140108.84k     170027.92k    179704.12k     182388.44k
aes-192 cbc     104770.95k     134601.12k     148900.05k    152662.30k     153941.11k
aes-256 cbc     95868.62k      116430.41k     124498.19k    127007.85k     127430.81k
sha256          23354.37k      54220.61k      99784.35k     126494.48k     138266.71k
sha512          16022.98k      64657.88k      113304.06k    178301.77k     214539.99k
```

最后，第三部分包含非对称加密算法的性能数据：

```
                  sign      verify     sign/s   verify/s
rsa   512 bits  0.000120s 0.000011s    8324.9    90730.0
rsa  1024 bits  0.000569s 0.000031s    1757.0    31897.1
rsa  2048 bits  0.003606s 0.000102s     277.3     9762.0
rsa  4096 bits  0.024072s 0.000376s      41.5     2657.4
                                   op      op/s
 160  bit ecdh  (secp160r1)   0.0003s    2890.2
 192  bit ecdh  (nistp192)   0.0006s    1702.9
 224  bit ecdh  (nistp224)   0.0006s    1743.5
 256  bit ecdh  (nistp256)   0.0007s    1513.3
 384  bit ecdh  (nistp384)   0.0015s     689.6
```

```
521  bit ecdh  (nistp521)   0.0029s    340.3
163  bit ecdh  (nistk163)   0.0009s   1126.2
233  bit ecdh  (nistk233)   0.0012s    818.5
283  bit ecdh  (nistk283)   0.0028s    360.2
409  bit ecdh  (nistk409)   0.0060s    166.3
571  bit ecdh  (nistk571)   0.0130s     76.8
163  bit ecdh  (nistb163)   0.0009s   1061.3
233  bit ecdh  (nistb233)   0.0013s    755.2
283  bit ecdh  (nistb283)   0.0030s    329.4
409  bit ecdh  (nistb409)   0.0067s    149.7
571  bit ecdh  (nistb571)   0.0146s     68.4
```

这些输出可以干什么？你可以对比相同平台下不同编译选项或者不同OpenSSL版本对性能的影响。例如前面的结果就是真实环境下的OpenSSL 0.9.8k运行之后的结果（发行方打过了补丁）。我迁移到OpenSSL 1.0.1h的出发点是因为希望支持TLS 1.1和TLS 1.2；那是否会有性能的影响呢？我下载编译了OpenSSL 1.0.1h进行测试。让我们看一下：

```
$ ./openssl-1.0.1h speed rsa
[...]
OpenSSL 1.0.1h 5 Jun 2014
built on: Thu Jul 3 18:30:06 BST 2014
options:bn(64,64) rc4(8x,int) des(idx,cisc,16,int) aes(partial) idea(int) blowfish(idx)
compiler: gcc -DOPENSSL_THREADS -D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H -Wa,--noexecstack -m64↩
-DL_ENDIAN -DTERMIO -O3 -Wall -DOPENSSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_ MONT5 ↩
-DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DMD5_ASM -DAES_ASM -DVPAES_ASM↩
-DBSAES_ASM -DWHIRLPOOL_ASM -DGHASH_ASM
                  sign    verify    sign/s  verify/s
rsa  512 bits 0.000102s 0.000008s   9818.0  133081.7
rsa 1024 bits 0.000326s 0.000020s   3067.2   50086.9
rsa 2048 bits 0.002209s 0.000068s    452.8   14693.6
rsa 4096 bits 0.015748s 0.000255s     63.5    3919.4
```

很显然，OpenSSL 1.0.1h的性能在这台服务器上几乎快一倍（使用2048位的RSA密钥）；性能从每秒277次签名操作提高到每秒450次。这个结果意味着我升级之后还能获得更好的性能，这真的是一个好消息！ 11

直接使用这个测试结果来衡量部署之后的性能并不合适，因为在实际场景中有很多影响性能的因素，而且很多影响因素与TLS没有直接关系（例如HTTP的长连接设置、缓存设置等）。因此这些数字最多可用来作一个粗略的估计。

在粗略估计之前，你还需要考虑一些东西。默认情况下speed命令只会使用一个进程。而现在大多数服务器都有多核，所以你必须让speed命令并发使用多个实例才能算出整台服务器能支持的TLS操作次数。使用-multi参数就可以实现这个测试，我的服务器有4个核，所以我使用下面这个命令：

```
$ openssl speed -multi 4 rsa
[...]
OpenSSL 0.9.8k 25 Mar 2009
built on: Wed May 23 00:02:00 UTC 2012
options:bn(64,64) md2(int) rc4(ptr,char) des(idx,cisc,16,int) aes(partial) blowfish(ptr2)
compiler: cc -fPIC -DOPENSSL_PIC -DZLIB -DOPENSSL_THREADS -D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H ↩
```

```
-m64-DL_ENDIAN -DTERMIO -O3 -Wa,--noexecstack -g -Wall -DMD32_REG_T=int -DOPENSSL_BN_ASM_MONT ↩
-DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DMD5_ASM -DAES_ASM
available timing options: TIMES TIMEB HZ=100 [sysconf value]
timing function used:
                  sign    verify    sign/s  verify/s
rsa  512 bits 0.000030s 0.000003s  33264.5  363636.4
rsa 1024 bits 0.000143s 0.000008s   6977.9  125000.0
rsa 2048 bits 0.000917s 0.000027s   1090.7   37068.1
rsa 4096 bits 0.006123s 0.000094s    163.3   10652.6
```

正如预期的那样，性能几乎是原来的4倍。我又看了一下每秒钟可以执行多少次RSA签名操作，因为这是一台服务器上最消耗CPU的密码操作，所以也是最先产生瓶颈的地方。1090次/秒的签名结果告诉我们，这台服务器每秒可以处理大约1000次全新的TLS连接。这对我来说已经足够了，而且有非常健康、安全的余量。因为我在服务器上启用了会话恢复，所以我知道能够每秒处理超过1000次的TLS连接。我希望这台服务器上的流量能够多到让我可以担心TLS的性能为止。

另外一个不能完全相信speed命令输出结果的原因在于，它不会默认使用可用密码中最快的实现方式。在某种情况下，默认的输出具有欺骗性。例如，支持AES-NI指令的服务器可以加速AES计算，在测试的时候这个特性不会默认被用到。

```
$ openssl speed aes-128-cbc
[...]
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
aes-128 cbc      67546.70k    74183.00k    69278.82k   155942.87k   156486.38k
```

为了激活硬件加速卡，需要在命令行上使用-evp开关：

```
$ openssl speed -evp aes-128-cbc
[...]
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
aes-128-cbc     188523.36k   223595.37k   229763.58k   203658.58k   206452.14k
```

11.4 创建私有证书颁发机构

如果想要建立自己的CA，OpenSSL已经包含了所有你需要的东西。所有的操作都通过纯命令行执行，虽然不那么友好，整个过程也比较长，但是这可以让你去思考每个细节。

我建议自己创建一套私有的CA主要是出于教学的目的，不过还有一些别的原因。OpenSSL的CA天然满足个人或者小团体的需求，例如在开发环境使用一套CA比到处使用自签名的证书好得多。同时还可以通过客户端证书来提供双向验证，这可以极大地提高敏感Web应用的安全性。

运行私有CA最大的挑战不是设置问题，而是如何保证基础结构的安全。例如根密钥必须离线保存，因为所有的安全都依赖它。另一方面，CRL和OCSP响应程序证书必须定期进行更新，而这会要求根密钥保持联机。

> **注意**
> 在你准备继续读下去的时候，我建议你先看一遍第3章，这样你会大致了解证书的结构以及证书颁发机构的运作。

11.4.1 功能和限制

在本节的剩余部分，我们会创建一个与公共CA类似的私有CA架构。会先有一个根CA，然后创建其他的二级CA。接着我们会通过CRL和OCSP服务提供证书吊销信息。为了让根CA的私钥可以离线保存，OCSP响应程序需要使用它们自己的身份。这并非是最简单的CA，但是相对来说比较安全。另外二级CA会在技术上进行限制，只能给允许的主机名签发证书。

完成设置之后，必须将根证书安全地分发给所有客户端。一旦根证书分发完毕，就可以开始签发客户端和服务器证书了。有一个限制是以这种方式设置的OCSP响应程序主要用来测试，因为只能承受比较小的负载。

11.4.2 创建根 CA

创建全新的CA有几个步骤：配置、创建目录结构和初始化密钥文件，最后生成根密钥和证书。本节描述了整个过程和常见的CA操作。

1. 根CA配置

创建CA之前，我们需要先准备一个配置文件告诉OpenSSL我们希望的配置。一般情况下并不需要配置文件，但是根CA的创建操作复杂，使用配置文件可以简便很多。OpenSSL的配置文件很强大，在开始之前我建议你熟悉一下这些配置的功能（命令行上使用`man config`命令）。

配置文件第一部分包括了CA的名称、基础URL和CA可分辨名称等基本信息。因为这些配置都很灵活，只需配置一次即可。

11

```
[default]
name                    = root-ca
domain_suffix           = example.com
aia_url                 = http://$name.$domain_suffix/$name.crt
crl_url                 = http://$name.$domain_suffix/$name.crl
ocsp_url                = http://ocsp.$name.$domain_suffix:9080
default_ca              = ca_default
name_opt                = utf8,esc_ctrl,multiline,lname,align

[ca_dn]
countryName             = "GB"
organizationName        = "Example"
commonName              = "Root CA"
```

第二部分直接控制了CA的操作。有关每个设置的完整信息，可以通过ca命令来获取它的文档（命令行上输入`man ca`）。大部分命令从字面意思就可以很容易理解，我们需要告诉OpenSSL存放文件的路径。因为根CA只用作二级CA的签发，所以我把有效期设置为10年。另外默认情况

下使用SHA256作为签名算法。

默认策略（policy_c_o_match）限制了这张CA签发的证书的国家名和组织名会与CA本身一样。对于公共CA来说很少会有这样的设置，但对于私有CA来说这种方式比较合适：

```
[ca_default]
home                    = .
database                = $home/db/index
serial                  = $home/db/serial
crlnumber               = $home/db/crlnumber
certificate             = $home/$name.crt
private_key             = $home/private/$name.key
RANDFILE                = $home/private/random
new_certs_dir           = $home/certs
unique_subject          = no
copy_extensions         = none
default_days            = 3650
default_crl_days        = 365
default_md              = sha256
policy                  = policy_c_o_match

[policy_c_o_match]
countryName             = match
stateOrProvinceName     = optional
organizationName        = match
organizationalUnitName  = optional
commonName              = supplied
emailAddress            = optional
```

第三部包含了req命令的配置，req命令只会在创建自签发根证书的时候用到一次。最重要的部分是扩展：基本限制（basicContraints）扩展表明这个证书是一张CA，密钥用法（keyUsage）扩展用来说明这个CA的用处：

```
[req]
default_bits            = 4096
encrypt_key             = yes
default_md              = sha256
utf8                    = yes
string_mask             = utf8only
prompt                  = no
distinguished_name      = ca_dn
req_extensions          = ca_ext

[ca_ext]
basicConstraints        = critical,CA:true
keyUsage                = critical,keyCertSign,cRLSign
subjectKeyIdentifier    = hash
```

配置的第四部分包括了根CA创建证书所需要的信息。因为基本限制（basicContraints）扩展的设置，所有的证书都将成为CA，但是我们需要把pathlen设置为0，表示这些CA无法再签发新的CA了。

所有二级CA都会受到限制，也就是说他们签发的证书只能对一些域名的子集有效，并且会

被限制使用场景。第一，扩展密钥用法（extendedKeyUsage）扩展限制了只能进行客户端验证（clientAuth）和服务器验证（serverAuth），也就是TLS的客户端和服务器验证。第二，名称限制（nameContraints）扩展限制了允许签发的域名只有example.com和example.org。理论上这样的设置让你可以签发二级CA给第三方，同时可以通过限制他们无法签发任意域名的主机名来保证安全。排除这两个IP段的要求来自CAB论坛的*Baseline Requirements*，该规范从技术上定义了对二级CA的限制。[①]

实际上，名称限制（nameContraints）并不完美，因为当前还有很多主流的平台无法识别名称限制扩展。如果你将这个扩展标记为关键扩展，就会导致很多平台拒绝识别你的证书。如果不将其标记为关键扩展，那么很多平台就不会识别这个扩展，导致名称限制实际没有任何效果。

```
[sub_ca_ext]
authorityInfoAccess      = @issuer_info
authorityKeyIdentifier   = keyid:always
basicConstraints         = critical,CA:true,pathlen:0
crlDistributionPoints    = @crl_info
extendedKeyUsage         = clientAuth,serverAuth
keyUsage                 = critical,keyCertSign,cRLSign
nameConstraints          = @name_constraints
subjectKeyIdentifier     = hash

[crl_info]
URI.0                    = $crl_url

[issuer_info]
caIssuers;URI.0          = $aia_url
OCSP;URI.0               = $ocsp_url

[name_constraints]
permitted;DNS.0=example.com
permitted;DNS.1=example.org
excluded;IP.0=0.0.0.0/0.0.0.0
excluded;IP.1=0:0:0:0:0:0:0:0/0:0:0:0:0:0:0:0
```

最后两部分的配置表示有了这个扩展的证书可以对OCSP响应进行签名。为了能够运行OCSP响应程序，我们生成一个特别的证书，并且将OCSP的签名能力赋予这张证书。从扩展可以看出这张证书不是一个CA：

```
[ocsp_ext]
authorityKeyIdentifier   = keyid:always
basicConstraints         = critical,CA:false
extendedKeyUsage         = OCSPSigning
keyUsage                 = critical,digitalSignature
subjectKeyIdentifier     = hash
```

2. 根CA的目录结构

下面我们会创建“根CA配置”中说到的目录结构，并且会初始化一些CA操作中会用到的

① *Baseline Requirements*，https://cabforum.org/baseline-requirements/（CAB论坛，检索于2014年7月9日）。

文件。

```
$ mkdir root-ca
$ cd root-ca
$ mkdir certs db private
$ chmod 700 private
$ touch db/index
$ openssl rand -hex 16 > db/serial
$ echo 1001 > db/crlnumber
```

我们会用到以下这几个目录。

- certs/
  存放证书的地方；证书在签名之后会放置到这个目录下。
- db/
  这个目录用于证书数据库（index），一些包括下一张证书以及CRL数字的文件。OpenSSL会创建额外需要的一些文件。
- private/
  这个目录会存放私钥，一个给CA使用，一个给OCSP响应程序使用。务必确保其他用户都不能访问这个目录（事实上，如果你真的很在意这个CA，那么这台存放根证书和密钥的服务器的用户账户必须尽可能少）。

> **注意**
> 创建一个新的CA证书的时候，就像本节我所做的这样，很重要的一点是使用随机数生成器来初始化证书序列号。当你使用同样的名称创建和发布多个CA证书的时候会非常有用（另外，常见的情况是你在生成证书的过程中出错了而不得不重新来过）；通过这样的方式可以让每个证书拥有不一样的序列号，这样就不会互相冲突了。

3. 根CA生成

我们需要分两步来创建根CA。首先，我们生成密钥和CSR文件。当我们使用-config开关之后，所有需要的信息都会从配置文件中加载进来：

```
$ openssl req -new \
    -config root-ca.conf \
    -out root-ca.csr \
    -keyout private/root-ca.key
```

第二步我们会创建自签名证书。-extension开关指向了配置文件的ca_ext部分，这样可以激活根CA所需的扩展。

```
$ openssl ca -selfsign \
    -config root-ca.conf \
    -in root-ca.csr \
    -out root-ca.crt \
    -extensions ca_ext
```

4. 数据库文件结构

db/index中的数据库是一个包含证书信息的明文文件，每行一个证书。我们刚刚创建根CA，

现在这个文件只有一行信息：

```
V    240706115345Z        1001    unknown    /C=GB/O=Example/CN=Root CA
```

每一行包括以下6个以制表符分隔的值。

(1) 状态标记［V表示有效（valid），R表示已吊销（revoked），E表示已过期（expired）］

(2) 过期时间（以YYMMDDHHMMSSZ格式表示）

(3) 吊销日期，如果没有被吊销则为空

(4) 序列号（十六进制）

(5) 文件路径（如果不知道就显示unknown）

(6) 可分辨名称

5. 根CA操作

使用ca命令的-gencrl开关给新CA生成CRL：

```
$ openssl ca -gencrl \
    -config root-ca.conf \
    -out root-ca.crl
```

使用ca的的命令来签发证书。需要注意的是-extensions开关需要指向配置文件里面正确的部分（例如，你肯定不希望再生成另一个根CA）。

```
$ openssl ca \
    -config root-ca.conf \
    -in sub-ca.csr \
    -out sub-ca.crt \
    -extensions sub_ca_ext
```

如果要吊销证书，可以使用ca命令的-revoke开关，不过需要有一份你想吊销的证书的副本。不过因为所有的证书都存在certs/目录下，所以只需要知道序列号即可。如果知道证书的可分辨名称，就可以在数据库里面查到它的序列号了。

11

为-crl_reason开关中的值选择一个正确的理由。该值可以是以下这些值之一：unspecified、keyCompromise、CACompromise、affiliationChanged、superseded、cessationOfOperation、certificateHold和removeFromCRL。

```
$ openssl ca \
    -config root-ca.conf \
    -revoke certs/1002.pem \
    -crl_reason keyCompromise
```

6. 创建用于OCSP签名的证书

首先我们需要给OCSP响应程序创建一个私钥和CSR。这两个操作对所有的非CA证书都适用，所以不需要指定配置文件：

```
$ openssl req -new \
    -newkey rsa:2048 \
    -subj "/C=GB/O=Example/CN=OCSP Root Responder" \
    -keyout private/root-ocsp.key \
    -out root-ocsp.csr
```

第二步需要使用根CA签发一张证书。-extensions开关的值选择ocsp_ext，以确保设置了OCSP签名所需要的扩展。我将这个证书的生命周期减少为365天(原来默认是3650天)。这些OCSP证书是没有吊销信息的，所以无法吊销它们。因此你会希望尽可能减少它们的生命周期。30天是一个比较好的选择，当然前提是你已经准备好频繁地创建新的OCSP证书。

```
$ openssl ca \
    -config root-ca.conf \
    -in root-ocsp.csr \
    -out root-ocsp.crt \
    -extensions ocsp_ext \
    -days 30
```

现在你已经有了OCSP响应程序所需要的一切东西，可以直接在根CA所在的服务器上进行测试。当然，如果在生产环境中使用，就必须将OCSP响应程序的密钥和证书放到别的地方：

```
$ openssl ocsp \
    -port 9080
    -index db/index \
    -rsigner root-ocsp.crt \
    -rkey private/root-ocsp.key \
    -CA root-ca.crt \
    -text
```

可以使用下面的命令来测试OCSP响应程序：

```
$ openssl ocsp \
    -issuer root-ca.crt \
    -CAfile root-ca.crt \
    -cert root-ocsp.crt \
    -url http://127.0.0.1:9080
```

输出结果中的verify OK表示已经成功验证签名，而good表示这张证书还没有被吊销。

```
Response verify OK
root-ocsp.crt: good
        This Update: Jul  9 18:45:34 2014 GMT
```

11.4.3　创建二级 CA

创建二级CA的过程和根CA几乎完全一样。本节我会突出那些不同之处，其他部分可以参考11.4.2节。

1. 二级CA配置

我们可以在前面根CA配置文件的基础上，进行一些适当的修改生成二级CA的配置。我们会把名称改为sub-ca并且使用另一个可分辨名称。我们将二级CA的OCSP响应程序放在另外一个端口，主要是因为ocsp命令不识别虚拟主机。如果为OCSP响应程序使用了适合的Web服务器，就可以完全避免使用特别的端口。该证书默认的生命周期是365天，我们会每隔30天生成全新的CRL。

将copy_extensions更改为copy意味着在生成新证书的时候，如果我们的配置文件里面没有设

置某些扩展，那么就会使用CSR文件里面的扩展字段。进行此更改之后，在准备CSR文件的时候就可以加入别的一些需要的字段，这些信息会在生成证书的时候加入到证书里面。这个特性有几分危险（因为允许其他人可以在一定程度上直接控制证书里面的内容），不过我认为在较小的环境中这么做是可以的。

```
[default]
name                  = sub-ca
ocsp_url              = http://ocsp.$name.$domain_suffix:9081

[ca_dn]
countryName           = "GB"
organizationName      = "Example"
commonName            = "Sub  CA"

[ca_default]
default_days          = 365
default_crl_days      = 30
copy_extensions       = copy
```

在配置文件的最后面，我们会增加两个新的配置，分别用于服务器和客户端证书的生成。唯一的区别就是keyUsage和extendedKeyUsage扩展。注意到我们把basicContraints扩展的值设置为false。之所以这么做，原因在于我们会复制CSR文件里面的扩展。如果在配置文件中没有显示设置这个扩展，那么就可能会用到CSR文件中的basicContraints了。

```
[server_ext]
authorityInfoAccess         = @issuer_info
authorityKeyIdentifier      = keyid:always
basicConstraints            = critical,CA:false
crlDistributionPoints       = @crl_info
extendedKeyUsage            = clientAuth,serverAuth
keyUsage                    = critical,digitalSignature,keyEncipherment
subjectKeyIdentifier        = hash

[client_ext]
authorityInfoAccess         = @issuer_info
authorityKeyIdentifier      = keyid:always
basicConstraints            = critical,CA:false
crlDistributionPoints       = @crl_info
extendedKeyUsage            = clientAuth
keyUsage                    = critical,digitalSignature
subjectKeyIdentifier        = hash
```

改好配置文件之后，按照根CA的过程创建一个同样的目录结构，不过可以使用另外一个名称，比如sub-ca。

2. 二级CA生成

与前面一样，创建二级CA需要两步。第一步生成密钥和CSR。当我们使用-config开关的时候，所有需要的信息都会从配置文件中加载进来。

```
$ openssl req -new \
```

11

```
    -config sub-ca.conf \
    -out sub-ca.csr \
    -keyout private/sub-ca.key
```

第二步我们使用根CA来签发证书。-extensions开关指向配置文件中的sub_ca_ext，从而使用二级CA所需要的扩展。

```
$ openssl ca \
    -config root-ca.conf \
    -in sub-ca.csr \
    -out sub-ca.crt \
    -extensions sub_ca_ext
```

3. 二级CA操作

要签发服务器证书，可以在处理CSR文件的时候，在-extensions开关中指定server_ext：

```
$ openssl ca \
    -config sub-ca.conf \
    -in server.csr \
    -out server.crt \
    -extensions server_ext
```

要签发客户端证书，可以在处理CSR文件的时候，在-extensions开关中指定client_ext：

```
$ openssl ca \
    -config sub-ca.conf \
    -in client.csr \
    -out client.crt \
    -extensions client_ext
```

> **注意**
>
> 当收到新证书申请请求的时候，你需要在对所有信息进行验证之后才能进行操作。你需要确保所有资料符合规定，特别是当你处理的CSR文件是别人生成的时。特别要注意证书的可分辨名称以及basicContraints和subjectAlternativeName扩展。

二级CA的CRL生成和证书的吊销过程与根CA是一样的。唯一不同的是OCSP响应程序所使用的端口；二级CA使用的是9081端口。推荐OCSP响应程序使用独立的证书，这样可以避免将二级CA部署到公开的服务器上。

第 12 章

使用OpenSSL进行测试

12

因为有大量的协议特性和实现上的一些怪癖，所以有时候很难确定安全服务器精确的配置和特性是什么。虽然现在有很多工具都适用于这个问题，但是很难知道它们是如何实现的，有时候很难完全信任它们的结果。尽管我花了很多年的时间测试安全服务器，也看到了很多好的工具，但是当我想要理解细节的时候，又会回归到使用OpenSSL和Wireshark。我并不是说每次测试都应该使用OpenSSL；相反，应该找到一种可信赖的自动化工具。在真的想要确定某些事情的时候，唯一的方式就是亲自使用OpenSSL去测试。

12.1 连接 SSL 服务

OpenSSL自带的客户端工具可以用来连接安全服务器。这个工具很像telnet或者nc，严格来说是处理SSL/TLS协议层，但是允许你完全控制其之上的协议层。

你需要提供主机名和端口来连接服务器，例如：

```
$ openssl s_client -connect www.feistyduck.com:443
```

一旦输入这个命令，就会看到很多诊断输出（稍后详细解释），然后你就有机会输入任何你需要的命令。因为我们访问的是一台HTTP服务器，所以最明智的做法是提交一个HTTP请求。下面的例子中，我提交一个HEAD请求，因为它告诉服务器不需要发送响应体。


```
HEAD / HTTP/1.0
Host: www.feistyduck.com

HTTP/1.1 200 OK
Date: Tue, 10 Mar 2015 17:13:23 GMT
Server: Apache
Strict-Transport-Security: max-age=31536000
Cache-control: no-cache, must-revalidate
Content-Type: text/html;charset=UTF-8
Transfer-Encoding: chunked
Set-Cookie: JSESSIONID=7F3D840B9C2FDB1FF7E5731590BD9C99; Path=/; Secure; HttpOnly
Connection: close

read:errno=0
```

现在我们知道TLS通信层在工作了：我们连上了HTTP服务器，提交了请求，然后收到了响

应。让我们回到诊断输出，最开始的几行显示的是服务器的证书信息。

```
CONNECTED(00000003)
depth=3 L = ValiCert Validation Network, O = "ValiCert, Inc.", OU = ValiCert Class ↩
2 Policy Validation Authority, CN = http://www.valicert.com/, emailAddress = ↩
info@valicert.com
verify error:num=19:self signed certificate in certificate chain
verify return:0
```

在我的系统上（你的系统也许也一样），s_client不会寻找系统默认的可信证书，它会提示在证书链中存在自签名证书。在大多数情况下不用关注证书是否有效，但是如果有必要的话，可以让s_client指定可信根证书，例如：

```
$ openssl s_client -connect www.feistyduck.com:443 -CAfile /etc/ssl/certs/ca-certificates.crt
CONNECTED(00000003)
depth=3 L = ValiCert Validation Network, O = "ValiCert, Inc.", OU = ValiCert Class 2 > Policy ↩
Validation Authority, CN = http://www.valicert.com/, emailAddress = info@valicert.com
verify return:1
depth=2 C = US, O = "Starfield Technologies, Inc.", OU = Starfield Class 2 Certification Authority
verify return:1
depth=1 C = US, ST = Arizona, L = Scottsdale, O = "Starfield Technologies, Inc.", OU = http://↩
certificates.starfieldtech.com/repository, CN = Starfield Secure Certification ↩
Authority, serialNumber = 10688435
verify return:1
depth=0 1.3.6.1.4.1.311.60.2.1.3 = GB, businessCategory = Private Organization, serialNumber =↩
06694169, C = GB, ST = London, L = London, O = Feisty Duck Ltd, CN = www.feistyduck.com
verify return:1
```

与之前提示的不一样，现在你可以看到它验证了证书链上每一级的证书。为了验证通过，你必须选择好CA证书路径。我在例子中使用的路径（/etc/ssl/certs/ca- certificates.crt）在Ubuntu 12.04 LTS上是有效的，但并不代表在你的系统上也是如此。如果不想使用系统提供的CA证书来进行验证，可以依赖由Mozilla提供的证书，我们曾经在11.1.4节中讨论过这些证书。

> **警告**
>
> Apple的操作系统OS X随机附带的OpenSSL经过修改，有时候会覆盖证书校验过程。换句话说，-CAfile开关可能无法如我们期望的那样工作。可以通过在调用s_client命令之前设置OPENSSL_X509_TEA_DISABLE环境变量来解决这个问题。[①]考虑到OS X上默认使用的OpenSSL还是基于非常陈旧的0.9.x分支，所以最好还是升级到最新的版本，可以使用Homebrew或者MacPorts进行安装。

输出中的下一部分按照服务器交付的顺序罗列了所有证书。

```
Certificate chain
0 s:/1.3.6.1.4.1.311.60.2.1.3=GB/businessCategory=Private Organization/serialNumber=06694169↩
/C=GB/ST=London/L=London/O=Feisty Duck Ltd/CN=www.feistyduck.com ↩
   i:/C=US/ST=Arizona/L=Scottsdale/O=Starfield Technologies, Inc./OU=http://certificates. ↩
```

① Apple OpenSSL Verification Surprises，https://hynek.me/articles/apple-openssl-verification-surprises/（Hynek Schlawack，2014年3月3日）。

```
starfieldtech.com/repository/CN=Starfield Secure Certification Authority/serialNumber=10688435
  1 s:/C=US/ST=Arizona/L=Scottsdale/O=Starfield Technologies, Inc./OU=http://certificates. ↩
starfieldtech.com/repository/CN=Starfield Secure Certification Authority/serialNumber=10688435
    i:/C=US/O=Starfield Technologies, Inc./OU=Starfield Class 2 Certification Authority
  2 s:/C=US/O=Starfield Technologies, Inc./OU=Starfield Class 2 Certification Authority
    i:/L=ValiCert Validation Network/O=ValiCert, Inc./OU=ValiCert Class 2 Policy Validation ↩
Authority/CN=http://www.valicert.com//emailAddress=info@valicert.com
  3 s:/L=ValiCert Validation Network/O=ValiCert, Inc./OU=ValiCert Class 2 Policy Validation ↩
Authority/CN=http://www.valicert.com//emailAddress=info@valicert.com
    i:/L=ValiCert Validation Network/O=ValiCert, Inc./OU=ValiCert Class 2 Policy Validation ↩
Authority/CN=http://www.valicert.com//emailAddress=info@valicert.com
```

每一张证书都会在第一行显示使用者信息，第二行显示颁发者信息。

这部分很重要，特别是当你需要看看到底发送了什么证书的时候；浏览器证书查看器一般都会显示重建的证书链，可能与上面呈现的会完全不一样。为了确定证书链是否正确，你可能会希望验证使用者和颁发者信息是否匹配。从最上面的分支（Web服务器）证书开始，逐级遍历列表，看看当前证书的颁发者与从属证书的使用者是否匹配。最后你看到的颁发者指向了某个不在证书链里面的根证书；或者是一个自己指向自己的根证书。

下一个输出的是服务器证书；它非常长，为了更好地呈现，我进行了简化：

```
Server certificate
-----BEGIN CERTIFICATE-----
MIIF5zCCBM+gAwIBAgIHBG9JXlv9vTANBgkqhkiG9w0BAQUFADCB3DELMAkGA1UE
[30 lines removed...]
os5LW3PhHz8y9YFep2SV4c7+NrlZISHOZVzN
-----END CERTIFICATE-----
subject=/1.3.6.1.4.1.311.60.2.1.3=GB/businessCategory=Private Organization /serialNumber=06694169 ↩
/C =GB/ST=London/L=London/O=Feisty Duck Ltd/CN=www.feistyduck.com
issuer=/C=US/ST=Arizona/L=Scottsdale/O=Starfield Technologies, Inc./OU=http://certificates. ↩
starfieldtech.com/repository/CN=Starfield Secure Certification Authority/serialNumber=10688435
```

> **注意**
>
> 任何时候当你在主题栏看到很长一串数字而不是一个名称的时候，意味着OpenSSL并不认识这个**对象标识符**。对象标识符是全球唯一而且是用来清楚指向某些东西的标识符。例如在前面的输出中，OID 1.3.6.1.4.1.311.60.2.1.3应该显示成jurisdictionOfIncorporationCountryName，这个OID用于**扩展验证**证书。

如果想深入了解证书，就需要先从输出内容中复制一份并保存在单独的文件中，我会在12.2节中具体讨论。

下面的信息是关于TLS连接的，意思从字面上就可以很容易理解了：

```
---
No client certificate CA names sent
---
SSL handshake has read 3043 bytes and written 375 bytes
---
New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-SHA
Server public key is 2048 bit
```

12

```
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol  : TLSv1.1
    Cipher    : ECDHE-RSA-AES256-SHA
    Session-ID: 032554E059DB27BF8CD87EBC53E9FF29376265F0BBFDBBFB7773D2277E5559F5
        Session-ID-ctx:
    Master-Key: 1A55823368DB6EFC397DEE2DC3382B5BB416A061C19CEE162362158E90F1FB0846EEFDB ↩
2CCF564A18764F1A98F79A768
    Key-Arg   : None
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 300 (seconds)
    TLS session ticket:
    0000 - 77 c3 47 09 c4 45 e4 65-90 25 8b fd 77 4c 12 da   w.G..E.e.%..wL..
    0010 - 38 f0 43 09 08 a1 ec f0-8d 86 f8 b1 f0 7e 4b a9   8.C.........~K.
    0020 - fe 9f 14 8e 66 d7 5a dc-0f d0 0c 25 fc 99 b8 aa   ....f.Z....%....
    0030 - 8f 93 56 5a ac cd f8 66-ac 94 00 8b d1 02 63 91   ..VZ...f......c.
    0040 - 05 47 af 98 11 81 65 d9-48 5b 44 bb 41 d8 24 e8   .G....e.H[D.A.$.
    0050 - 2e 08 2d bb 25 59 f0 8f-bf aa 5c b6 fa 9c 12 a6   ..-.%Y....\.....
    0060 - a1 66 3f 84 2c f6 0f 06-51 c0 64 24 7a 9a 48 96   .f?.,...Q.d$z.H.
    0070 - a7 f6 a9 6e 94 f2 71 10-ff 00 4d 7a 97 e3 f5 8b   ...n..q...Mz....
    0080 - 2d 1a 19 9c 1a 8d e0 9c-e5 55 cd be d7 24 2e 24   -........U...$.$
    0090 - fc 59 54 b0 f8 f1 0a 5f-03 08 52 0d 90 99 c4 78   .YT...._..R....x
    00a0 - d2 93 61 d8 eb 76 15 27-03 5e a4 db 0c 05 bb 51   ..a..v.'.^.....Q
    00b0 - 6c 65 76 9b 4e 6b 6c 19-69 33 2a bd 02 1f 71 14   lev.Nkl.i3*...q.

    Start Time: 1390553737
    Timeout   : 300 (sec)
    Verify return code: 0 (ok)
---
```

最重要的信息是协议版本（TLS 1.1）和使用的密码套件（ECDHE-RSA-AES256-SHA）。另外还能发现服务器返回了一个会话ID和TLS 会话票证（一种在非服务器维护状态下即可恢复会话的方式），而且也支持安全重新协商。一旦理解了这些输出内容所包含的信息，以后就无需再关注它了。

> **警告**
> 操作系统发行版常常自带工具并且与普通版本都有所差别。这里还有另外一个例子：前面一个命令协商了TLS 1.1，即便服务器支持TLS 1.2，为什么？原来是这样的，Ubuntu 12.04 TLS附带的某些OpenSSL版本禁用了TLS 1.2作为客户端连接，为的是避免某些互操作性问题。为了避免这类问题，我建议你使用之前配置和编译过的OpenSSL版本。

12.2　测试升级到 SSL 的协议

当使用HTTP的时候，TLS将整个明文通讯通道包起来成为HTTPS。其他一些协议开始是明

文，然后升级到加密模式。如果想要测试这类协议，那么就需要告诉OpenSSL是什么协议，这样OpenSSL就能够代表你进行升级。在-startssl开关后面带上协议信息。例如：

```
$ openssl s_client -connect gmail-smtp-in.l.google.com:25 -starttls smtp
```

在撰写本书的时候，OpenSSL支持smtp、pop3、imap、ftp和xmpp。

12.3 使用不同的握手格式

有时用OpenSSL测试服务器的时候，即便你知道服务器支持TLS（例如，使用浏览器访问的时候TLS可以正常工作），你对服务器的通信也可能会失败。其中一个可能的因素是服务器不支持老版本的SSL 2握手。

因为OpenSSL会尝试使用它理解的所有协议去进行协商，而且SSL 2默认情况下只能用老的SSL 2握手去进行协商。虽然这是个非常古老而且不安全的协议版本，但是从技术上来说老的握手过程并不是不安全的。它支持版本升级，也就是说可以协商更好的协议。然而SSL 2的握手格式不支持其后设计的许多连接协商特性。

因此，如果遇到了无法正常工作的情况，而且无法准确判断是什么的话，可以强制OpenSSL使用更新的握手格式。可以通过禁用SSL 2来实现：

```
$ openssl s_client -connect www.feistyduck.com:443 -no_ssl2
```

另外一种实现同样效果的方式是在命令行上指明要访问的主机名：

```
$ openssl s_client -connect www.feistyduck.com:443 -servername www.feistyduck.com
```

为了指明主机名，OpenSSL需要使用一种更新的握手协议里面的特性（该特性叫作服务器名称指示，server name indication或SNI），该特性会强制废弃老的格式。

12.4 提取远程证书

当使用s_client连接远程安全服务器的时候，它会将服务器的PEM编码格式证书转储到标准输出。如果需要证书的话，那么可以从回滚（scroll-back）缓冲区里面复制出来。如果提前知道你只需要获得证书的话，那么可以使用下面这个命令：

```
$ echo | openssl s_client -connect www.feistyduck.com:443 2>&1 | sed --quiet '/-BEGIN ↩
CERTIFICATE-/,/-END CERTIFICATE-/p' > www.feistyduck.com.crt
```

最前面的echo命令的目的是将shell命令和s_client分开。如果不用echo的话，那么s_client会一直等待你的输入，直到服务器超时（可能需要很长一段时间）。

默认情况下s_client只打印分支证书；如果希望打印完整的证书链，请使用-showcerts开关。使用了这个开关，前面这条命令会将所有的证书输出到同一个文件里面。

12.5　测试支持的协议

默认情况下s_client会尝试使用最高级的协议和远程服务器进行通信，并且在输出内容里面报告协商的版本信息。

```
Protocol  : TLSv1.1
```

有两种方式可以测试服务器是否某些版本的协议。第一种是在-ssl2、-ssl3、-tls1、-tls1_1或者是-tls1_2开关中明确地选择一个协议进行测试。另外一种方法是在-no_ssl2、-no_ssl3、-no_tls1、-no_tls1_1或者-no_tls1_2中选择一个或者多个你不想要进行测试的协议。

> **注意**
> 不是所有的OpenSSL都支持所有版本的协议。例如，老版本的OpenSSL不支持TLS 1.1和TLS 1.2，而新版本则可能不支持老的SSL 2之类的协议。

例如，当测试服务器不支持某个协议版本的时候可能会得到下面这些输出信息：

```
$ openssl s_client -connect www.example.com:443 -tls1_2 CONNECTED(00000003)
140455015261856:error:1408F10B:SSL routines:SSL3_GET_RECORD:wrong version number:s3_pkt.c:340:
---
no peer certificate available
---
No client certificate CA names sent
---
SSL handshake has read 5 bytes and written 7 bytes
---
New, (NONE), Cipher is (NONE)
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : 0000
    Session-ID:
    Session-ID-ctx:
    Master-Key:
    Key-Arg   : None
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    Start Time: 1339231204
    Timeout   : 7200 (sec)
    Verify return code: 0 (ok)
---
```

12.6　测试支持的密码套件

如果你想要用OpenSSL去确定远程服务器是否支持某个特殊密码套件，这里有一个小技巧。

加密配置字符串的设计是为了选择你想要使用的套件，所以如果只指定一个套件并且与服务器握手成功，那么就可以确定服务器支持这个套件。如果握手失败了，你就知道它并不支持。

我们以一个例子来测试一下服务器是否支持RC4-SHA算法，键入以下命令：

```
$ openssl s_client -connect www.feistyduck.com:443 -cipher RC4-SHA
```

如果想确定特定服务器支持的所有算法套件，可以通过调用openssl cipehrs ALL来获取你的OpenSSL版本支持的所有套件列表，然后一个个提交给服务器来测试服务器是否支持该密码套件。我并非让你手动去做这些测试，通过一点点的自动化程序就可以很好地解决这个问题。事实上，遇到这种情况的时候去网上找个好工具可能是一个更好的办法。

这种测试方法有一个缺点就是你只能测试你的OpenSSL支持的密码套件。如果用的是1.0之前的版本，就会是一个大问题，因为那些版本支持的套件数量非常少（例如，我的服务器使用的0.9.8k版本只支持32个套件）。从1.0.1分支开始的版本都支持超过100种套件，几乎是能用到的所有套件了。

没有一种SSL/TLS库支持所有的密码套件，所以想要进行综合测试就有点困难了。在SSL Labs，进行部分握手测试时会遇到这种情况，我是通过使用定制的客户端去假装支持任意密码套件的方式来支持的。该客户端实际上甚至无法完成任何一个套件的协商，但是只需要发出协商就已经可以让我们知道服务器是否支持某个套件。使用这种方式不仅可以测试所有的套件，而且效率非常高。

12.7 测试要求包含 SNI 的服务器

最开始SSL和TLS协议设计的时候，每个IP端点（IP地址加端口）只支持一个网站。TLS的SNI扩展是为了在同一个IP端点上支持多张证书。TLS客户端使用这个扩展发送希望访问的主机名，而TLS服务器使用该扩展来选择的要用以响应的正确证书。简单来说，SNI支持了虚拟安全托管。

因为很多服务器还不支持SNI，所以大部分情况下你在使用s_client的时候不需要指定主机名。但是当你遇到启用了SNI支持的系统，就会遇到下面三种情况中的任意一种。

- 大多数情况下，无论是否提供SNI信息都会获得同样的证书。
- 服务器返回的证书可能是别的站点的证书，不是你想要测试的证书。
- 极少情况下服务器可能会中止握手，并且拒绝连接。

你可以在s_client中使用-servername开关来启用SNI：

```
$ openssl s_client -connect www.feistyduck.com:443 -servername www.feistyduck.com
```

你可以通过使用和不使用SNI开关，并且检查证书是否相同来确定某个站点是否支持SNI。如果证书不相同，表示SNI是必需的。

有时候，如果请求的主机名不可用，服务器会返回TLS警告。即便就服务器而言，这个警告不是致命的，客户端还是可能决定关闭连接。例如，使用老版本的OpenSSL（例如1.0.0之前的版本），你会得到下面的错误信息：

```
$ /opt/openssl-0.9.8k/bin/openssl s_client -connect www.feistyduck.com:443 -servername xyz.com
CONNECTED(00000003)
1255:error:14077458:SSL routines:SSL23_GET_SERVER_HELLO:reason(1112):s23_clnt.c:596:
```

12.8　测试会话复用

当加上-reconnect开关的时候，s_client命令可以用来测试会话复用。在此模式下，s_client与目标服务器连接6次，在第一次的时候会创建新的会话，然后在接下来的5次请求中复用同样的会话：

```
$ echo | openssl s_client -connect www.feistyduck.com:443 -reconnect
```

上面这个命令会产生大量输出，大部分都无需理会。最重要的部分是新会话和重复会话的信息。这些信息中应该只会在开头部分有一个新的会话，如下面这行：

```
New, TLSv1/SSLv3, Cipher is RC4-SHA
```

之后是5次复用的会话，如下面这样：

```
Reused, TLSv1/SSLv3, Cipher is RC4-SHA
```

大部分情况下你可能不会关心所有的输出，想要快点知道答案。那么可以使用下面这个命令：

```
$ echo | openssl s_client -connect www.feistyduck.com:443 -reconnect -no_ssl2 2>/dev/null | grep ↩
'New\|Reuse'
New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Reused, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Reused, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Reused, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Reused, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Reused, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
```

下面是该命令的功能介绍。

- -reconnect开关激活了会话复用模式。
- -no_ssl2开关表明我们不希望使用SSL 2连接，这样第一次连接就会使用SSL 3以及更高的版本。老的SSL 2握手格式不支持TLS扩展，如果遇到支持会话票证作为会话复用的服务器就会出现问题。
- 2> /dev/null部分将你不需要关心的标准错误隐藏掉了。
- 最后，通过管道化的grep命令过滤出你关心的那几行。

> **注意**
> 如果在测试会话复用的时候不想使用会话票证，例如有时候不是所有客户端都支持该特性，可以使用-no_ticket开关禁用会话票证。

12.9　检查 OCSP 吊销状态

如果OCSP响应程序出现故障，有时候很难理解到底是为什么。用命令行检测证书吊销状态

是可行的，但并不是非常直接。需要执行下面这些步骤。

(1) 获得想要检查吊销状态的证书。

(2) 获得颁发证书。

(3) 确定OCSP响应程序的URL。

(4) 提交OCSP请求并观察响应。

前面两个步骤可以在连接服务器的时候指明-showcerts开关：

```
$ openssl s_client -connect www.feistyduck.com:443 -showcerts
```

输出信息的第一张证书属于服务器证书。如果证书链的配置没错的话，第二张证书就是颁发者的证书。为了确保没有问题，需要检查第一张证书的颁发者与第二张证书的使用者是否匹配：

```
---
Certificate chain
 0 s:/1.3.6.1.4.1.311.60.2.1.3=GB/businessCategory=Private Organization/serialNumber=06694169/C ↩
=GB/ST=London/L=London/O=Feisty Duck Ltd/CN=www.feistyduck.com
   i:/C=US/ST=Arizona/L=Scottsdale/O=Starfield Technologies, Inc./OU=http://certificates. ↩
starfieldtech.com/repository/CN=Starfield Secure Certification Authority/serialNumber=10688435
-----BEGIN CERTIFICATE-----
MIIF5zCCBM+gAwIBAgIHBG9JXlv9vTANBgkqhkiG9w0BAQUFADCB3DELMAkGA1UE
[30 lines of text removed]
os5LW3PhHz8y9YFep2SV4c7+NrlZISHOZVzN
-----END CERTIFICATE-----
 1 s:/C=US/ST=Arizona/L=Scottsdale/O=Starfield Technologies, Inc./OU=http://certificates. ↩
starfieldtech.com/repository/CN=Starfield Secure Certification Authority/serialNumber= ↩
10688435
   i:/C=US/O=Starfield Technologies, Inc./OU=Starfield Class 2 Certification Authority
-----BEGIN CERTIFICATE-----
MIIFBzCCA++gAwIBAgICAgEwDQYJKoZIhvcNAQEFBQAwaDELMAkGA1UEBhMCVVMx
[...]
```

如果第二张证书不对的话，需要检测证书链里剩余的证书；有些服务器配置了错误的证书链顺序。如果无法在证书链中找到颁发者的证书，那就得去别的地方找找。有一种方式是通过查找分支证书的颁发机构信息访问（authority information access）扩展：

```
 $ openssl x509 -in fd.crt -noout -text
[...]
    Authority Information Access:
        OCSP - URI:http://ocsp.starfieldtech.com/
        CA Issuers - URI:http://certificates.starfieldtech.com/repository/sf_intermediate.crt
[...]
```

如果看到CA颁发者（CA issuer）的信息，就应该能找到颁发者证书的URL。如果颁发者的信息不存在，那么可以尝试在浏览器中打开这个网站，让浏览器构建整个证书链，然后从浏览器的证书查看器里面下载颁发者的证书。如果以上所有的尝试都失败的话，可以在可信库里面进行查找或者访问CA的网站。

如果你已经有了证书，想知道OCSP响应程序的地址，可以使用x509命令带上-ocsp_uri开关：

```
$ openssl x509 -in fd.crt -noout -ocsp_uri
http://ocsp.starfieldtech.com/
```

现在可以提交OCSP请求了：

```
$ openssl ocsp -issuer issuer.crt -cert fd.crt -url http://ocsp.starfieldtech.com/-CAfile ↩
issuer.crt
WARNING: no nonce in response
Response verify OK
fd.crt: good
        This Update: Feb 18 17:59:10 2013 GMT
        Next Update: Feb 18 23:59:10 2013 GMT
```

你需要在响应里面查找两个结果。第一，检查响应自身是否有效（上面这个例子是Response Verify OK表示有效），第二，检查响应的结果是什么。如果你看到的状态是good那么表示该证书没有被吊销，如果是revoked则表示该证书已经被吊销了。

> **注意**
>
> 警告信息里面说的缺少nonce（加密通信中仅使用一次的密钥）的意思是OpenSSL想要使用nonce来防止重放攻击，但是服务器的响应中并没有nonce。这种情况多数是因为CA希望提高他们的OCSP响应程序的性能。当他们禁用了nonce保护（标准允许这么做），OCSP响应能够被制造（一般是批量制造）、缓存并且在一段时间内重用。

你可能会遇到某些OCSP响应程序无法成功响应前面的命令。这种情况下，下面的建议可能会有所帮助。

- 不要请求nonce

  有些服务器无法很好地处理nonce，响应就会出错。默认情况下OpenSSL会请求一个nonce，如果要禁用，可以通过在命令中指定-no_nonce开关。

- 在请求头中提供主机名

  有些HTTP请求没有在Host头中指定正确的主机名，虽然大部分的OCSP服务器都能正常响应，但也有一些不行。如果你遇到的错误里面包含HTTP错误码（例如404），那么可以尝试在OCSP请求的时候带上主机名。在OpenSSL 1.0.0以及更高版本中可以使用-header开关，不过该开关无法在文档中查到。

有了前面这两个要点，最终可以使用下面这行命令：

```
$ openssl ocsp -issuer issuer.crt -cert fd.crt -url http://ocsp.starfieldtech.com/ -CAfile issuer. ↩
crt -no_nonce -header Host ocsp.starfieldtech.com
```

12.10　测试 OCSP stapling

OCSP stapling是一个可选特性，它允许在传输服务器证书的时候带上OCSP响应来验证证书的有效性。因为OCSP响应是通过已经存在的连接进行传输的，所以客户端无需另外单独获取。

只有客户端在握手请求的时候提交status_request扩展，服务器才会使用OCSP stapling。服务器如果支持OCSP stapling的话，就会在握手里面带上OCSP响应信息。

使用s_client工具带上-status 开关来请求OCSP stapling：

```
$ echo | openssl s_client -connect www.feistyduck.com:443 -status
```

OCSP相关信息会在连接输出的最开始展现出来。例如，如果服务器不支持stapling的话，在输出的最开始部分就会看到：

```
CONNECTED(00000003)
OCSP response: no response sent
```

如果服务器支持stapling，那么会在输出内容中看到完整的OCSP响应：

```
OCSP Response Data:
    OCSP Response Status: successful (0x0)
    Response Type: Basic OCSP Response
    Version: 1 (0x0)
    Responder Id: C = US, O = "GeoTrust, Inc.", CN = RapidSSL OCSP-TGV Responder
    Produced At: Jan 22 17:48:55 2014 GMT
    Responses:
    Certificate ID:
      Hash Algorithm: sha1
      Issuer Name Hash: 834F7C75EAC6542FED58B2BD2B15802865301E0E
      Issuer Key Hash: 6B693D6A18424ADD8F026539FD35248678911630
      Serial Number: 0FE760
    Cert Status: good
    This Update: Jan 22 17:48:55 2014 GMT
    Next Update: Jan 29 17:48:55 2014 GMT
[...]
```

证书状态为good表示该证书未被吊销。

12.11 检查 CRL 吊销状态

与OCSP相比，通过检查证书吊销列表（certificate revocation list，CRL）来检测证书有效性会更复杂，过程如下所示。

(1) 获得你想要检测吊销状态的证书。

(2) 获得颁发证书。

(3) 下载并验证CRL。

(4) 在CRL列表中查找证书序列号。

第一步与OCSP检测方式一样，可以按照12.9节中的步骤实施。

CRL的地址已经编码进服务器证书，可以使用下面的命令取出：

```
$ openssl x509 -in fd.crt -noout -text | grep crl
                URI:http://rapidssl-crl.geotrust.com/crls/rapidssl.crl
```

从CA获取CRL：

```
$ wget http://rapidssl-crl.geotrust.com/crls/rapidssl.crl
```

验证CRL是有效的（例如，是由颁发者的证书签名的）：

```
$ openssl crl -in rapidssl.crl -inform DER -CAfile issuer.crt -noout
verify OK
```

现在，确定希望进行检测的证书的序列号：

```
$ openssl x509 -in fd.crt -noout -serial
serial=0FE760
```

此时，可以将CRL转换成可读的的格式，然后手动检查：

```
$ openssl crl -in rapidssl.crl -inform DER -text -noout
Certificate Revocation List (CRL):
        Version 2 (0x1)
    Signature Algorithm: sha1WithRSAEncryption
        Issuer: /C=US/O=GeoTrust, Inc./CN=RapidSSL CA
        Last Update: Jan 25 11:03:00 2014 GMT
        Next Update: Feb  4 11:03:00 2014 GMT
        CRL extensions:
            X509v3 Authority Key Identifier:
                keyid:6B:69:3D:6A:18:42:4A:DD:8F:02:65:39:FD:35:24:86:78:91:16:30

            X509v3 CRL Number:
                92103
Revoked Certificates:
    Serial Number: 0F38D7
        Revocation Date: Nov 26 20:07:51 2013 GMT
    Serial Number: 6F29
        Revocation Date: Aug 15 20:48:57 2011 GMT
[...]
    Serial Number: 0C184E
        Revocation Date: Jun 13 23:00:12 2013 GMT
    Signature Algorithm: sha1WithRSAEncryption
         95:df:e5:59:bc:95:e8:2f:bb:0a:4f:20:ad:ca:8f:78:16:54:
         35:32:55:b0:c9:be:5b:89:da:ba:ae:67:19:6e:07:23:4d:5f:
         16:18:5c:f3:91:15:da:9e:68:b0:81:da:68:26:a0:33:9d:34:
         2d:5c:84:4b:70:fa:76:27:3a:fc:15:27:e8:4b:3a:6e:2e:1c:
         2c:71:58:15:8e:c2:7a:ac:9f:04:c0:f6:3c:f5:ee:e5:77:10:
         e7:88:83:00:44:c4:75:c4:2b:d3:09:55:b9:46:bf:fd:09:22:
         de:ab:07:64:3b:82:c0:4c:2e:10:9b:ab:dd:d2:cb:0c:a9:b0:
         51:7b:46:98:15:83:97:e5:ed:3d:ea:b9:65:d4:10:05:10:66:
         09:5c:c9:d3:88:c6:fb:28:0e:92:1e:35:b0:e0:25:35:65:b9:
         98:92:c7:fd:e2:c7:cc:e3:b5:48:08:27:1c:e5:fc:7f:31:8f:
         0a:be:b2:62:dd:45:3b:fb:4f:25:62:66:45:34:eb:63:44:43:
         cb:3b:40:77:b3:7f:6c:83:5c:99:4b:93:d9:39:62:48:5d:8c:
         63:e2:a8:26:64:5d:08:e5:c3:08:e2:09:b0:d1:44:7b:92:96:
         aa:45:9f:ed:36:f8:62:60:66:42:1c:ea:e9:9a:06:25:c4:85:
         fc:77:f2:71
```

在CRL的开头是一些元数据，之后是吊销证书的列表，结尾部分是签名（我们在上一步中验证过）。如果服务器证书的序列号在这份列表里面，意味着该证书已经被吊销了。

如果不想逐行寻找序列号（有一些CRL可能会很长），那就用grep查找它，但是需要小心，确保格式正确（例如，如有必要，可以移除最开始的0x，删除所有在开头的0，并且将所有字母变成大写）。例如：

```
$ openssl crl -in rapidssl.crl -inform DER -text -noout | grep FE760
```

12.12 测试重新协商

s_client工具还有几个可以协助你手动进行重新协商测试的特性。首先，当你连接的时候，s_client会报告远程服务器是否支持重新协商。因为服务器如果支持安全重新协商的话，那么它会在握手阶段通过交换特殊的TLS扩展来表明支持该特性。如果服务器支持，输出可能是这样的（我已将重点加粗）：

```
New, TLSv1/SSLv3, Cipher is AES256-SHA
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
SSL-Session:
    [...]
```

如果服务器不支持安全重新协商，输出会有一些不同：

```
Secure Renegotiation IS NOT supported
```

即便服务器表明它支持安全重新协商，你也想测试一下他是否允许客户端开始这个重新协商。由客户端发起的重新协商（client-initiated renegotiation）是一个在实际中用不到的协议特性（因为如果有必要的话都可以由服务器发起重新协商），该特性会使得服务器更容易受到拒绝式服务攻击。

要开始重新协商，需要在单独一行中输入R字符。例如，假设我们正在与HTTP服务器通信，你可以在第一行发送一个请求，初始化重新协商，然后结束请求。在向一个支持客户端发起重新协商的Web服务器发起请求的时候，会出现以下内容：

```
HEAD / HTTP/1.0
R
RENEGOTIATING
depth=3 C = US, O = "VeriSign, Inc.", OU = Class 3 Public Primary Certification Authority
verify return:1
depth=2 C = US, O = "VeriSign, Inc.", OU = VeriSign Trust Network, OU = "(c) 2006 VeriSign, Inc. ↩
- For authorized use only", CN = VeriSign Class 3 Public Primary Certification Authority - G5
verify return:1
depth=1 C = US, O = "VeriSign, Inc.", OU = VeriSign Trust Network, OU = Terms of use at ↩
https://www.verisign.com/rpa (c)06, CN = VeriSign Class 3 Extended Validation SSL CA
verify return:1
depth=0 1.3.6.1.4.1.311.60.2.1.3 = US, 1.3.6.1.4.1.311.60.2.1.2 = California, businessCategory = ↩
Private Organization, serialNumber = C2759208, C = US, ST = California, L = Mountain View, O ↩
= Mozilla Corporation, OU = Terms of use at www.verisign.com/rpa (c)05, OU = Terms of use at ↩
www.verisign.com/rpa (c)05, CN = addons.mozilla.org
verify return:1
Host: addons.mozilla.org

HTTP/1.1 301 MOVED PERMANENTLY
Content-Type: text/html; charset=utf-8
Date: Tue, 05 Jun 2012 16:42:51 GMT
Location: https://addons.mozilla.org/en-US/firefox/
Keep-Alive: timeout=5, max=998
```

```
Transfer-Encoding: chunked
Connection: close

read:errno=0
```

当发生了重新协商，服务器会重新将它的证书发送给客户端。你可以在输出内容中看到对证书链的校验过程，在那之后是Host请求头。能够看到Web服务器的响应就证明服务器是支持重新协商的。由于在不同版本的SSL/TLS库中发现了不同形式的重新协商问题，不支持重新协商的服务器可能会直接断开连接，或者即使保持连接处于打开状态，也会拒绝在该连接上继续通信（结果通常是超时）。

不支持重新协商的服务器会直截了当地拒绝在该连接上进行第二次握手：

```
HEAD / HTTP/1.0
R
RENEGOTIATING
140003560109728:error:1409E0E5:SSL routines:SSL3_WRITE_BYTES:ssl handshake failure:s3_pkt.c:592:
```

撰写本书的时候，OpenSSL默认是按照服务器不支持安全重新协商的方式进行连接的。它也支持安全和不安全的重新协商，由服务器作出选择。如果与一个不支持安全重新协商的服务器进行重新协商并且成功，表明该服务器支持由客户端发起的不安全重新协商。

> **注意**
> 测试不安全重新协商最可靠的方式是采用本节使用的方法，但是要使用那些有重新协商功能的OpenSSL版本（例如，0.9.8k）。我提及这个的主要原因是有部分服务器同时支持安全和不安全的重新协商，而现代版本的OpenSSL只支持安全重新协商选项，导致很难检测该漏洞。

12.13　测试 BEAST 漏洞

BEAST攻击利用TLS 1.1之前所有SSL和TLS协议的漏洞。该漏洞影响所有CBC套件以及客户端和服务器的数据流；然而BEAST攻击只对客户端有效。现代浏览器使用被称为1/*n*−1分割的工作区来阻止被利用，不过一些服务器仍然在它们这一端部署减缓措施，特别是如果它们的一些用户还使用老的浏览器（未打补丁）。

理想的解决方式是使用TLS 1.1以及更高版本，但是支持这些新协议的客户端还不够多。事实上RC4本身现在也被认为不安全，这导致情况变得更加复杂。如果你认为BEAST比RC4更危险，那么应该为使用新浏览器的用户部署TLS 1.2，对其他所有人使用RC4。

- 严格缓解

  如果使用TLS 1.0及之前的协议，那么不要使用任何CBC套件，只启用RC4套件。不支持RC4的客户端无法协商安全的连接，这种模式排除了网站潜在的一些用户，但是一些PCI的评审员对此有要求。

- RC4优先

因为只有很小一部分客户端不支持RC4，第二种方式是启用CBC套件，但是对所有支持RC4的客户端强制使用RC4套件。这种方式为大部分用户提供了保护。

你对服务器行为的期望决定了如何进行测试。在两种方式下，通过使用-no_ssl2、-no_tls_1_1以及-no_tls_1_2开关来确保只使用了不安全的协议。

为了测试严格缓解的方式，可以尝试让客户端在连接的时候禁用所有的RC4套件：

```
$ echo | openssl s_client -connect www.feistyduck.com:443 \
-cipher 'ALL:!RC4' -no_ssl2 -no_tls1_1 -no_tls1_2
```

如果连接成功了（只有在使用了有漏洞的CBC套件时才有可能），你就可以确定严格缓解的方案并没有生效。

为了测试RC4优先的方式，可以尝试在连接的时候将RC4放在密码套件列表的最后面：

```
$ echo | openssl s_client -connect www.feistyduck.com:443 \
-cipher 'ALL:+RC4' -no_ssl2 -no_tls1_1 -no_tls1_2
```

支持RC4优先的服务器会为连接选择其中一种RC4套件，并且忽略所有其他CBC套件。如果再看看别的，可以发现服务器并没有使用任何缓解BEAST方式。

12.14 测试心脏出血

你可以手动测试心脏出血，也可以使用一些工具（因为心脏出血漏洞非常容易利用，所以有非常多的工具），不过这类工具的准确性都存在疑问。一些证据表明有些工具无法成功检测有漏洞的服务器。[①]考虑到心脏出血的严重性，最好手动测试或者使用那些可以让你看到整个过程的工具。我会用一个修改过的OpenSSL版本来描述你可以使用的一种方式。

如果你的OpenSSL版本支持心跳协议的话（1.0.1及更高版本），测试的某些部分不需要对OpenSSL进行修改。例如，如果要测试远程服务器是否支持心跳协议，请在连接的时候使用-tlsextdebug开关显示服务器扩展：

```
$ openssl s_client -connect www.feistyduck.com:443 -tlsextdebug
CONNECTED(00000003)
TLS server extension "renegotiation info" (id=65281), len=1
0001 - <SPACES/NULS>
TLS server extension "EC point formats" (id=11), len=4
0000 - 03 00 01 02                                    ....
TLS server extension "session ticket" (id=35), len=0
TLS server extension "heartbeat" (id=15), len=1
0000 - 01
[...]
```

不支持心跳扩展的服务器不会受到心脏出血漏洞的影响。要测试服务器是否会响应心跳请求，使用-msg开关请求将该协议消息展示出来，然后连接服务器，输入B并按回车：

① Bugs in Heartbleed detection scripts，https://www.hut3.net/blog/cns---networks-security/2014/04/14/bugs-in-heartbleed-detection-scripts-（Shannon Simpson和Adrian Hayter，2014年4月14日）。

```
$ openssl s_client -connect www.feistyduck.com:443 -tlsextdebug -msg
[...]
---
B
HEARTBEATING
>>> TLS 1.2  [length 0025], HeartbeatRequest
    01 00 12 00 00 3c 83 1a 9f 1a 5c 84 aa 86 9e 20
    c7 a2 ac d7 6f f0 c9 63 9b d5 85 bf 9a 47 61 27
    d5 22 4c 70 75
<<< TLS 1.2  [length 0025], HeartbeatResponse
    02 00 12 00 00 3c 83 1a 9f 1a 5c 84 aa 86 9e 20
    c7 a2 ac d7 6f 52 4c ee b3 d8 a1 75 9a 6b bd 74
    f8 60 32 99 1c
read R BLOCK
```

输出内容显示了一对完整的心跳请求和响应。两条心跳消息的第二和第三字节是载荷（payload）的长度。我们提交了18字节（十六进制为12）的载荷，然后服务器返回了同样长度的载荷。两个例子中还有额外的16字节的填充。载荷的开始两字节组成了序列号，是OpenSSL用来匹配请求以及响应的。剩余的载荷以及填充部分都是随机数据。

要测试服务器是否有漏洞，我们需要准备一个特别的OpenSSL版本，它可以发送不正确的载荷长度。有漏洞的服务器会接受攻击者宣称的载荷长度，并且会以同样字节的内容作出响应，而不管攻击者是否真的提供了这么长的载荷。

此时你要决定是否想要创建具有攻击性的测试（通过获取进程的数据来攻击服务器）或者非攻击性的测试，这取决于你的环境。如果你有测试的许可，那么就采取攻击性测试。在这种情况下，你可以清楚地看到返回的内容，而且不会出现误判。例如，某些版本的GnuTLS支持心跳而且会响应带有不正确载荷长度的请求，但是却不会返回服务器数据。非攻击性测试无法可靠地检测出这种情况。

下面针对OpenSSL 1.0.1h的补丁形成了一个非攻击性测试的版本：

```
--- t1_lib.c.original   2014-07-04 17:29:35.092000000 +0100
+++ t1_lib.c    2014-07-04 17:31:44.528000000 +0100
@@ -2583,6 +2583,7 @@
 #endif

 #ifndef OPENSSL_NO_HEARTBEATS
+#define PAYLOAD_EXTRA 16
int
tls1_process_heartbeat(SSL *s)
        {
@@ -2646,7 +2647,7 @@
               /* 序列号 */
                n2s(pl, seq);

-               if (payload == 18 && seq == s->tlsext_hb_seq)
+               if ((payload == (18 + PAYLOAD_EXTRA)) && seq == s->tlsext_hb_seq)
                        {
                        s->tlsext_hb_seq++;
                        s->tlsext_hb_pending = 0;
```

```
@@ -2705,7 +2706,7 @@
        /*  消息类型 */
        *p++ = TLS1_HB_REQUEST;
        /* 载荷长度 (18字节)  */
-       s2n(payload, p);
+       s2n(payload + PAYLOAD_EXTRA, p);
        /* 序列号 */
        s2n(s->tlsext_hb_seq, p);
        /* 16随机字节 */
```

要建立非攻击性测试，将载荷长度增加到16字节或者将填充的长度增加到16字节。漏洞服务器对这样的请求作出响应时，将返回填充而不会返回任何其他内容。要建立攻击性测试，则将载荷长度增加到32字节。漏洞服务器会以50字节的载荷（OpenSSL默认发送的18字节加上你的32字节）作出响应并发送16字节的填充信息。通过使用这种方式增加发送载荷的长度，漏洞服务器最终会发送64 KB的数据。对心脏出血免疫的服务器则不会进行响应。

为了建立你自己的心脏出血测试工具，解压缩一个全新的OpenSSL源代码，请编辑`ssl/t1_lib.c`，按照补丁进行修改，然后像平时一样进行编译，但是不要进行安装。最终的`openssl`二进制会被放在apps/子目录中。因为采用的是静态编译，你可以将它重命名为类似`openssl-heartbleed`，然后移动到某个固定的地方。

下面是一个输出的例子，你可以获得漏洞服务器返回的16字节服务器数据（以粗体显示）：

```
B
HEARTBEATING
>>> TLS 1.2 [length 0025], HeartbeatRequest
    01 00 32 00 00 7c e8 f5 62 35 03 bb 00 34 19 4d
    57 7e f1 e5 90 6e 71 a9 26 85 96 1c c4 2b eb d5
    93 e2 d7 bb 5f
<<< TLS 1.2 [length 0045], HeartbeatResponse
    02 00 32 00 00 7c e8 f5 62 35 03 bb 00 34 19 4d
    57 7e f1 e5 90 6e 71 a9 26 85 96 1c c4 2b eb d5
    93 e2 d7 bb 5f 6f 81 0f aa dc e0 47 62 3f 7e dc
    60 95 c6 ba df c9 f6 9d 2b c8 66 f8 a5 45 64 0b
    d2 f5 3d a9 ad
read R BLOCK
```

如果你想要在一个响应中看到更多的数据，可以增加载荷的长度，重新编译，然后再测试一遍。也可以通过再次输入B命令，获取同样长度的另外一批数据。

12.15 确定 Diffie-Hellman 参数的强度

在OpenSSL 1.0.2以及之后的版本中，当你在连接服务器的时候，如果使用了临时Diffie-Hellman，`s_client`命令会输出密钥的强度。因此，要确定某些服务器DH参数的强度，你所要做的就是，在连接服务器的时候只提供那些将DH用作密钥交换的套件。例如：

```
$ openssl-1.0.2 s_client -connnect www.feistyduck.com:443 -cipher kEDH
[...]
---
```

12

```
No client certificate CA names sent
Peer signing digest: SHA512
Server Temp Key: DH, 2048 bits
---
[...]
```

支持出口套件的那些服务器可能提供了更加不安全的DH参数。为了检测这种情况，在连接的时候只提供出口DHE套件：

```
$ openssl-1.0.2 s_client -connnect www.feistyduck.com:443 -cipher kEDH+EXPORT
```

对那些正确配置的服务器，这条命令会失败。否则你会发现服务器会协商不安全的512位的DH参数。

第 13 章

配置Apache

13

Apache httpd是一款流行的Web服务器，一经发布就被大量网站使用。Apache是一个成熟的产品，其2.4.x分支对TLS有极好的支持，特别是在最近的版本中（显著的改进是在2.4.7版本）。如果是源代码编译Apache，你可以利用所有可用的功能。

在实践中，大多数人有机会接触2.2.x分支的一些版本，因为这是前几代流行的服务器版本（例如Debian、Ubuntu、Red Hat Enterprise Linux等）所配备的。目前这代服务器已经或者即将配备Apache 2.4.x，这意味着新版本会慢慢开始流行。表13-1列出了2.2.x和2.4.x分支的主要差异。

表13-1 Apache httpd的各个最新稳定分支对TLS功能的支持

	Apache 2.2.x	Apache 2.4.x
DH参数默认强度	支持（2.2.31）[a]	支持（2.4.7）[b]
可配置DH和ECDH参数	支持（2.2.31）[a]	支持（2.4.7）[b]
椭圆曲线支持	支持（2.2.26）[c]	支持
OCSP stapling	–	支持
分布式TLS会话缓存	–	支持
可配置会话票证密钥	–	支持
禁用会话票证	–	支持（2.4.12）[d]

a 2015年7月发布的版本2.2.31。
b 2013年11月发布的版本2.4.7。
c 2013年11月发布的版本 2.2.26。早期版本可以通过一个叫作TLS Interposer的第三方工具（本章稍后将介绍）支持ECDH密钥交换。
d 2015年1月发布的版本2.4.12。

13

> **注意**
> 大多数操作系统发行版配备相同（或类似）版本号的软件包，但与官方原始版本在功能上有所不同。这些差异最常见是安全补丁，但也可能是功能差异。你应该检查文档和源代码（软件包通常包含原始的源代码和补丁），以了解差异是否重要。

2.2.x分支实际上最大的问题是缺乏对椭圆曲线加密的支持。虽然Apache在2.2.26（2013年11月发布）中增加了对EC的支持，但大多数发行版所配备的支持都基于一些早期的版本。不支持EC密码就无法部署ECDH密钥交换，这意味着你不能具备快速并且强大的前向保密支持。一些发

行版向后移值了重要特性；请检查你自己的版本是否存在这种可能。

缺乏其他特性是可以容忍的。OCSP stapling是值得拥有的（它可以提高网站性能），但对大多数人来说并不是非常重要。如果是你认为的重要特性，可以从源代码安装Apache 2.4.x。

除了重大明显的差异，2.4.x分支包含大量的微改进。虽然一开始并不明显，但它们累加起来就可能会有显著的效果。作为一个例子，Apache 2.4.x很可能占用更少的内存，因为它使用了OpenSSL减少内存消耗模式（`SSL_MODE_RELEASE_BUFFERS`选项）。我检查时发现，OpenSSL的这个特性并没有在最新的2.2.x版本中启用。

本章旨在涵盖关于Apache的TLS配置最重要和有趣的方面，但并不是参考指南。有关更多细节，请参考官方文档。

13.1　安装静态编译 OpenSSL 的 Apache

早在2004年我在写第一本关于Apache安全的书（*Apache Security*）时，源代码安装Apache就是很常见的，我花了很多时间记录这个过程。随着技术的稳定，大多数人停止了使用源代码安装（因为容易被依赖操作系统提供的二进制文件所困扰）。你的操作系统供应商发布安全更新的速度可能比你从源代码编译更快，因此也更加安全。

现在，要使用最好的TLS特性，我们有时不得不挽起袖子回到老旧的方式。例如，我有两台运行Ubuntu10.04 LTS的服务器，上面安装的OpenSSL版本不支持TLS 1.2，并且安装的Apache 2.2.x也不支持ECDHE密码套件。[①]

如果你正在运行较旧的发行版，使用最新版本的OpenSSL运行Apache最简单的方法是静态编译加密库代码，全部安装到一个单独的位置。这样一来，你既可以达到目的又不会影响到操作系统的其余部分。

首先，获得OpenSSL的最新稳定版本，并在不会影响你的系统版本的位置进行安装。按照11.1.2节中的说明进行安装。

然后，取得Apache、APR和APR-Util库的最新版本。将所有软件包解压到同一个源代码树下，并将后两者放在Apache期望的位置：

```
$ tar zxvf httpd-2.4.16.tar.gz
$ cd httpd-2.4.16/srclib/
$ tar zxvf ../../apr-1.5.2.tar.gz
$ ln -s apr-1.5.2/ apr
$ tar zxvf ../../apr-util-1.5.4.tar.gz
$ ln -s apr-util-1.5.4/ apr-util
```

你现在可以配置和安装Apache了。`mod_ssl`这个模块将被静态编译；所有其他模块将被动态编译。

```
$ ./configure \
```

① Ubuntu 10.04 LTS现在已经过时了，但在2014年8月本书第一版发布时仍然是受支持的。如果你在阅读这篇文章时仍在运行旧版本的Ubuntu，我建议你尽快升级。

```
    --prefix=/opt/httpd \
    --with-included-apr \
    --enable-ssl \
    --with-ssl=/opt/openssl-1.0.1p\
    --enable-ssl-staticlib-deps \
    --enable-mods-static=ssl
$ make
$ sudo make install
```

这里你可以继续调整配置。所有模块默认将被编译，但只有一些模块将在配置中启用。

13.2 启用 TLS

如果要在HTTPS默认端口（443）上部署网站，Apache会考虑对IP地址自动启用TLS协议。你需要显式启用TLS的唯一情形是，使用非标准端口时。例如：

```
# 443端口默认启用
Listen 192.168.0.1:443

# 但是其他非443端口需要显式启用
Listen 192.168.0.1:8443 https
```

你会发现许多协议配置并不使用Listen指令，而是使用SSLEngine指令启用TLS：

```
<VirtualHost 192.168.0.1:443>
    # 网站主机名
    ServerName site1.example.com

    # 对虚拟主机启用TLS
    SSLEngine on

    # 其他配置指令
    ...
</VirtualHost>
```

这种做法从Apache 2.0.x开始很受使用者欢迎，因为在这些版本中的Listen指令没有针对协议配置的支持。

> **注意**
> Apache实现了Web服务器和代理服务器。因此，也有控制这两个角色操作TLS的配置指令。大多数代理指令以SSLProxy开头；在配置Web服务器端时应该忽略它们。

13

13.3 配置 TLS 协议

配置Apache前端TLS需要三个指令。首先是SSLProtocol，指定哪些协议应该启用：

```
# 启用全部协议，除了SSL 2和 SSL 3这两个过时并且不安全的协议
SSLProtocol all -SSLv2 -SSLv3
```

通常的做法是通过all启用所有可用的协议，然后禁用你不希望部署的。第二个指令是

SSLHonorCipherOrder，指示Apache在TLS握手期间选择其首选套件（而不是选择客户端所提供的第一个支持套件）：

```
# 由服务器选择密码套件，而不是客户端
SSLHonorCipherOrder on
```

最后SSLCipherSuite指令需要一个OpenSSL的套件配置字符串以及配置这些套件以何种顺序被启用：

```
# 这份密码套件配置只使用提供前向保密的，并且以最佳性能排序
SSLCipherSuite "ECDHE-ECDSA-AES128-GCM-SHA256 \
ECDHE-ECDSA-AES256-GCM-SHA384 \
ECDHE-ECDSA-AES128-SHA \
ECDHE-ECDSA-AES256-SHA \
ECDHE-ECDSA-AES128-SHA256 \
ECDHE-ECDSA-AES256-SHA384 \
ECDHE-RSA-AES128-GCM-SHA256 \
ECDHE-RSA-AES256-GCM-SHA384 \
ECDHE-RSA-AES128-SHA \
ECDHE-RSA-AES256-SHA \
ECDHE-RSA-AES128-SHA256 \
ECDHE-RSA-AES256-SHA384 \
DHE-RSA-AES128-GCM-SHA256 \
DHE-RSA-AES256-GCM-SHA384 \
DHE-RSA-AES128-SHA \
DHE-RSA-AES256-SHA \
DHE-RSA-AES128-SHA256 \
DHE-RSA-AES256-SHA256 \
EDH-RSA-DES-CBC3-SHA"
```

> **注意**
> 上述例子中的密码套件配置是安全的，但根据自己的偏爱和风险状况，可能你喜欢的套件配置略有不同。你可以在第8章中找到关于TLS服务器配置的详细论述，并可以在11.3.1节的“推荐配置”部分中找到部署OpenSSL的例子。

前面的例子主要是针对较新的Apache版本，其中有椭圆加密支持，但在较老的安装版本中会适当地回退。

> **提示**
> TLS协议配置最好放置在主服务器范围内，它适用于托管在服务器上的所有网站。仅在必要时调整每个网站的基础配置。

13.4　配置密钥和证书

除了配置TLS协议，安全的网站还需要私钥和证书链。为此，通常需要三个指令，如下面的例子所示：

```
# 配置服务器私钥
SSLCertificateKeyFile conf/server.key
```

```
# 配置服务器证书
SSLCertificateFile conf/server.crt

# 配置CA提供的中间证书链，当服务器是自签名证书时不需要这个指令
SSLCertificateChainFile conf/chain.pem
```

> **注意**
> 从2.4.8版本开始SSLCertificateChainFile指令已经过时了。取代它的是通过SSLCertificateFile指令指定提供所有的证书文件。这种变化可能是因为越来越多的网站要使用多私钥部署（例如同时使用RSA和ECDSA），并且每个私钥可能需要不同的证书链这一事实驱动的。

没有正确配置整个证书链是一个常见的错误，导致连接的客户端证书警告。为了避免这个问题，始终遵循你的CA提供的说明。当更新证书时，请确保使用新的中间证书；旧的中间证书可能不再合适。

> **注意**
> 本节中的示例假定你的私钥不使用密码保护。我建议密钥创建和备份时使用密码，但在服务器上部署时不使用密码。如果你想使用保护的密钥，将不得不使用SSLPassPhraseDialog指令让Apache与外部程序对接，每次需要时提供密码。

13.5　配置多个密钥

Apache允许安全的网站使用不止一种类型的TLS的密钥，这并不广为人知。这个特性最初被设计为允许网站并行部署RSA和DSA密钥，它几乎是无人使用的，因为DSA在Web服务器密钥中消失殆尽。

近期出现大量为了提高握手性能而部署ECDSA私钥的讨论。与此同时由于目前广泛使用的SHA1已经接近寿终正寝，已有的证书签名向SHA2迁移。问题是旧客户端可能不支持ECDSA密钥和SHA2签名。一种解决方案是部署两套密钥和证书：将RSA和SHA1部署给旧客户端，而将ECDSA和SHA2部署给新客户端。

部署有多个私钥的网站非常简单：只需指定多个密钥和证书，一个接着另一个。例如：

```
# RSA私钥
SSLCertificateKeyFile conf/server-rsa.key
SSLCertificateFile conf/server-rsa.crt

# DSA私钥
SSLCertificateKeyFile conf/server-dsa.key
SSLCertificateFile conf/server-dsa.crt

# ECDSA私钥
SSLCertificateKeyFile conf/server-ecdsa.key
SSLCertificateFile conf/server-ecdsa.crt
```

```
# 中间证书必须同时适用三种服务器证书
SSLCertificateChainFile conf/chain.pem
```

唯一要注意的是每个服务器只能使用一次SSLCertificateChainFile指令，这意味着三个证书的中间证书必须相同。早期开始提供ECDSA密钥的CA都设置了这种方式。

使用不同的证书层次结构是可能的，但必须完全避免SSLCertificateChainFile。相反，将所有必要的中间证书（对应所有密钥的）串联到一个文件中，并使用SSLCertificateFile指令指向它。使用这种方法可能会有轻微的性能降低，因为对每一个新的连接，OpenSSL都需要检查可用的CA证书，才能构建证书链。

如果将Apache 2.4.8或更高版本与OpenSSL 1.0.2或更高版本一起使用，就完全支持多个独立的证书层次结构。在这种情况下，简单地将SSLCertificateFile指令指向包含整个证书链的文件，证书链以分支证书为开始按正确的顺序排列。

> **注意**
> 为了确保所有部署的密钥确实可用，请确保在配置中启用了相应的密码套件。ECDSA套件在名称中都有ECDSA；DSA套件在名称中都有DSS；所有其他套件都被设计成与RSA密钥一起工作。

13.6　通配符和多站点证书

如果有两个或多个站点共用一个证书，在相同IP地址部署是可行的，尽管事实上对公众网站的虚拟安全托管还不可行。不需要特殊的配置；所有这些简单关联的网站使用相同的IP地址，并确保它们都使用相同的证书。①

这能够工作是因为TLS终止和HTTP主机的选择是两个独立的步骤。终止TLS时，如果缺少SNI信息（更多信息参见13.7节），那么Apache就会为该IP地址提供第一个出现在配置中的默认站点的证书。第二步，Apache查看提供的Host请求头，并在HTTP级别上提供正确的站点。如果未在IP地址上配置请求的主机名，将提供默认的网站。

使用这种类型的部署，你可能会得到类似下面这样的警告：

```
[Mon Dec 30 11:26:04.058505 2013] [ssl:warn] [pid 31136:tid 140679275079488] AH0229↩
2: Init: Name-based SSL virtual hosts only work for clients with TLS server name ↩
indication support (RFC 4366)
```

这是因为Apache注意到你在同一端点有多个安全的站点，但它不会检查默认证书对所有站点是否有效。从2.4.10版本开始，不再显示警告。

① 从技术上讲，该限制是每IP地址和端口组合（一个TCP/IP端点）。例如，你可以将一个安全站点托管在192.168.0.1:443上，而将另一个托管在192.168.0.1.:8443上。在实践中，仅可以将公共站点托管在端口443（HTTPS的标准端口）上，因此限制是每IP地址有效的。

13.7 虚拟安全托管

与13.6节中讨论的设置不同，真正的虚拟安全托管有许多不相关的网站，每一个都有自己的证书，共享一个IP地址。因为SSL和TLS的早期版本不支持这个功能，所以会有很多客户端不支持。出于这个原因，将虚拟安全托管用于用户广泛的公共网站还不可行，但可以将其用于现代用户群的站点。

Apache支持虚拟安全托管，并在需要时自动使用它。唯一的问题是：如果你依赖虚拟安全托管，但收到不支持的客户端，此时会发生什么？通常，在这种情况下Apache会提供与所请求的IP地址关联的默认站点的证书。因为该证书不可能与所需的主机名匹配，用户便以证书警告结束。但是，如果它们能够绕过警告，就会得到它们希望看到的站点。[①]

你不能避免这种情况下的证书警告，但最好的做法依赖虚拟安全托管的网站对不明白SNI客户不提供任何内容。这就是SSLStrictSNIVHostCheck指令做的事，使用它有两种方式。

第一种方式是对整个IP地址执行严格的虚拟安全托管。要做到这一点，你要在默认虚拟主机上使用该指令。例如：

```
# Apache 2.2.x需要下面的指令才能支持基于名称的虚拟托管
# Apache 2.4.x以及更高版本不需要
NameVirtualHost 192.168.0.1:443

<VirtualHost 192.168.0.1:443>
    ServerName does-not-exist.example.com

    #对不支持虚拟安全托管（通过SNI）的客户端不提供任何内容
    SSLStrictSNIVHostCheck On

    ...
</VirtualHost>

<VirtualHost 192.168.0.1:443>
    ServerName site1.example.com
...
</VirtualHost>

<VirtualHost 192.168.0.1:443>
    ServerName site2.example.com
    ...
</VirtualHost>
```

或者你只对某些站点执行严格的虚拟安全托管，而对其他站点进行宽松的配置。在下面的例子中，不会将site1.example.com提供给不支持SNI的客户端，但会提供其他站点：

```
<VirtualHost 192.168.0.1:443>
    ServerName default.example.com
    ...
</VirtualHost>
```

① 假设所请求的主机名在服务器上已经配置；如果不是，它们将再次得到默认的网站。

```
<VirtualHost 192.168.0.1:443>
    ServerName site1.example.com

    # 该网站对不支持虚拟安全托管（通过SNI）的客户端不提供任何内容
    SSLStrictSNIVHostCheck On

    ...
</VirtualHost>

<VirtualHost 192.168.0.1:443>
    ServerName site2.example.com
    ...
</VirtualHost>
```

每当因为严格检查SNI发生错误，Apache将返回状态码403以迫使请求失败，但不会指示根本原因。如果主机头中提供的信息是正确的，匹配主机的ErrorDocument指令会进行协商。如果它指定重定向或消息，那么该消息将被发送回客户端。如果ErrorDocument指定文件或脚本，它的处理将会失败。

如果想对这种情况提供一条自定义错误消息，可以这样做：通过禁用内置SNI严格检查并且实现自定义检查来代替。Apache变量SSL_TLS_SN包含客户端提供的SNI信息；如果这个变量是空的，意味着客户端不支持SNI。下面的mod_rewrite配置（放置在某个虚拟主机部分中）可以工作：

```
RewriteEngine On
RewriteCond %{SSL:SSL_TLS_SNI} =""
RewriteRule ^ /errors/no-sni.html
```

> **注意**
>
> 这里所描述的行为在版本2.4.9中开始实现。从版本2.4.10开始，Apache的行为有所不同：(1) 403响应页面包含了拒绝的原因；(2) ErrorDocument指令可以调用脚本。这些改变使得有可能配置脚本来处理403错误，检测错误提示是否提及SNI（REDIRECT_ERROR_NOTES变量），并根据具体的上下文提供不同的信息。

13.8　为错误消息保留默认站点

对未正确指定的请求提供实际的网站内容响应从来就不是一个好主意。例如，你不希望搜索引擎对任意主机名的网站进行索引。无论你提供什么内容，客户端都会将其视为属于它请求的站点；不匹配有时可以用于利用一个站点中的漏洞，就好像该漏洞在另一个站点上一样。为了避免这种情况，我建议保留每个IP地址和端口组合的默认站点用于传递错误消息。

下面是可以使用的配置示例：

```
# 我们将要使用此默认网站向用户
# 解释主机不匹配和SNI问题
<VirtualHost 192.168.0.1:443>
    # 此处使用的主机名永远不应该被匹配
    ServerName does-not-exist.example.com
```

```
    DocumentRoot /var/www/does-not-exist

    # 这个IP和端口上的全部网站都需要支持SNI
    SSLStrictSNIVHostCheck on

    # 到这个网站的全部请求返回404状态码
    RewriteEngine On
    RewriteRule ^ - [L,R=404]

    # 对请求未在此服务器上进行配置的
    # 主机名的客户端返回的错误信息
    ErrorDocument 404 "<h1>No such site</h1><p>The site you requested does not exist.</p>"

    # 根据需要使用的其他配置指令
    # 像往常一样启用TLS，并使用自签名证书
    ...
</VirtualHost>
```

13.9 前向保密

如果你是从2.4.x的分支部署Apache，并且从源代码编译一切，那么就可以自己处置DHE和ECDHE套件，这些套件为前向保密提供了可靠的支持。如果依赖系统提供的软件包，它们有时会不支持EC加密，原因如下。

- EC加密不受老版本的Apache 2.2.x支持

  即使使用了配套的OpenSSL版本，很多流行的Apache 2.2.x发行版本并不支持EC加密。这主要是因为OpenSSL增加了EC支持，但在默认情况下该功能是禁用的。如果你面临这种情况，那么有可能会不希望从源代码安装Apache（我会在本节稍后进行解释）。

- OpenSSL版本太老

  如果底层OpenSSL不支持较新的功能（如EC加密），那么即使Apache支持也无能为力。较早的OpenSSL版本依然广泛用于旧设备上，甚至一些较新的操作系统版本也使用。例如，在2013年11月发布的OS X Mavericks，附带OpenSSL 0.9.8y（这是早期0.9.x分支的最新版本）。

  当前最好用的OpenSSL版本是从1.0.1分支取得最新的或更高的版本。幸运的是，Apache可以用静态编译OpenSSL的版本，这意味着你可以只升级Web服务器同时不影响操作系统的核心软件包。

- OpenSSL的版本不支持EC

  长期以来，Red Hat的操作系统自带的OpenSSL不支持 EC加密，因为他们的律师希望在EC加密成为可信任的专利后再使用。这让使用Fedora和Red Hat Enterprise Linux发行版（及其开源衍生物，如CentOS）的人难以部署前向保密。[1]处理它的唯一办法是重新编译关键

13

[1] ECDHE是很重要的，因为唯一可替代它的DHE套件在Internet Explorer上无法实现向前保密。最重要的是，DHE比RSA和ECDHE密钥交换要慢得多，这就是为什么大多数网站不希望使用它。

的系统软件包。

这一情况在2013年10月发生了改变，当时Fedora18及其更高版本更新了启用EC加密的OpenSSL版本。①

从2013年11月发布的6.5版本开始，所有的Red Hat Enterprise Linux版本都支持EC加密。②

> **以非补丁方式在Apache 2.2.x中启用ECDHE套件**
>
> TLS Interposer③是一个Linux工具，用于改善程序对OpenSSL的使用，使其不必重新编译或以任何方式进行改变。它的工作原理是拦截一定的OpenSSL函数调用并且重写它们的行为。
>
> 默认情况下，TLS Interposer将执行以下操作。
>
> - 禁用SSL 2和SSL 3协议
> - 启用ECDHE密码套件支持
> - 执行它自己的密码套件配置，默认是使用强密码加密的
>
> 一个不错的TLS Interposer使用案例是在Apache 2.2.x中启用ECDHE密码套件。此工具无法将所有EC功能添加到Apache中，但增加ECDHE套件使你能够支持强大的前向保密，这是最常见的需求。

13.10　OCSP stapling

在线证书状态协议（online certificate status protocol，OCSP）是按需取得证书撤销信息的协议。大多数证书包含OCSP信息，这使得TLS客户端能够直接与签发CA通信确认证书没有被吊销。OCSP stapling允许Web服务器从CA获取新的OCSP响应，缓存在本地，并与证书一起提交给客户端。在这种情况下，客户端不需要与CA通信；这提高了性能并且有更好的隐私。Apache从2.4.x分支开始支持OCSP stapling。

13.10.1　配置 OCSP stapling

虽然Apache有很多与OCSP stapling相关的指令，但大多只需要进行微调。你一开始启用OCSP stapling只需要以下两个指令：

```
# 基于服务器上使用的证书数量，
# 配置OCSP响应缓存大小128 KB
SSLStaplingCache shmcb:/opt/httpd/logs/stapling_cache(128000)
```

---

① Bug#319901: missing ec and ecparam commands in openssl package，https://bugzilla.redhat.com/show_bug.cgi?id=319901（Red Hat Bugzilla，2013年10月22日关闭）。

② Red Hat Enterprise Linux6.5 Release Notes，https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html-single/6.5_Release_Notes/（Red Hat，2013年11月21日）。

③ TLS Interposer，https://netfuture.ch/tools/tls-interposer/（Marcel Waldvogel，检索于2014年7月12日）。

```
# 为服务器上的全部站点
# 默认启用OCSP stapling
SSLUseStapling on
```

在这个例子中，我配置了OCSP响应的服务器级缓存，默认对所有站点启用stapling。你也可以在别处使用SSLUseStapling指令为单个网站启用或禁用stapling。

默认情况下，会将成功的OCSP响应缓存3600秒，但可以使用SSLStaplingStandardCacheTimeout指令改变这个超时时间。

> **注意**
> OCSP请求是基于HTTP的，这意味着需要允许你的Web服务器出站，在互联网上请求各种OCSP响应程序。如果正在运行出站防火墙，确认允许该通信例外。

如果你的网站没有正确配置证书链，配置OCSPstapling可能会失败。为了通过Apache来验证OCSP响应（它总是如此），需要签发服务器证书的CA证书；如果没有，stapling无法工作并且Apache会报错：

```
[Thu Jan 23 16:26:58.547877 2014] [ssl:error] [pid 1333:tid 140576489142080] AH0221↩
7: ssl_stapling_init_cert: Can't retrieve issuer certificate!
[Thu Jan 23 16:26:58.547900 2014] [ssl:error] [pid 1333:tid 140576489142080] AH0223↩
5: Unable to configure server certificate for stapling
```

如果由于某种原因你没有使用SSLCertificateChainFile配置证书链，可以在SSLCertificateFile指令中配置所需的CA证书。其实，最好的做法是始终有根证书在证书链。

要使用OpenSSL验证OCSP stapling配置是否正确，请按照12.10节中的说明操作。

13.10.2 处理错误

Apache会缓存成功和失败OCSP响应。理论上这没有坏处，因为你的客户端通过直接与CA对话也会获得相同的结果。但在实践中，这取决于实际情况。例如，因为即使是失败的响应也会被缓存（默认情况下为600秒，通过SSLStaplingErrorCacheTimeout可修改），一次性的问题可能最终会影响到所有的用户。

由于有很多的传闻证据表明，OCSP响应程序可能有奇怪的问题，我想你应该谨慎行事，而不是返回响应程序错误：

```
SSLStaplingReturnResponderErrors off
```

如果你选择传播错误，要记住在真正的OCSP响应程序不响应的情况下，Apache会默认生成虚假的OCSP tryLater响应。我认为禁用此功能比较安全：

```
SSLStaplingFakeTryLater off
```

这可能是出问题的一个例子，考虑别人重新配置你的Web服务器出站防火墙，无意阻止Apache取得OCSP响应。如果禁用虚假响应，你的客户将仍然能够直接与真正的响应者进行沟通。

13.10.3　使用自定义 OCSP 响应程序

通常情况下，OCSP请求提交到证书中列出的OCSP响应程序。但在以下两种情况下你可能想硬编码OCSP响应程序信息。

- 某些证书实际可能不包含任何OCSP信息，即使签发CA提供了响应程序。这种情况下可以手动提供OCSP响应程序地址。
- 在严格安全管控的环境中，从Web服务器的直接出站通信可能会被禁止。在这种情况下，如果想使用OCSP stapling，你需要为OCSP请求配置HTTP代理。

可以使用`SSLStaplingForceURL`指令在全局或单个站点覆盖证书OCSP信息：

```
SSLStaplingForceURL http://ocsp.example.com
```

13.11　配置临时的 DH 密钥交换

传统上，Apache保留使用OpenSSL默认配置的Diffie-Hellman（DH）密钥交换强度。这样运作了很长一段时间，但OpenSSL默认的1024位强度不再被认为是足够的。相比之下，目前最佳实践的服务器密钥至少需要2048位。在很长一段时间，增加DH密钥交换强度的唯一途径是修改源代码，使用一个只对2.4.x分支可用的补丁。[①]但没有其他办法。从2.4.7版本开始，Apache会自动调整DH密钥交换的强度来匹配相应私钥的强度。

> **警告**
>
> 如果你坚持使用旧版本的Apache和1024位的DH密钥交换，需要确定这对你的情况而言是否太脆弱。在一些已知攻击中（参考6.5节），Apache中硬编码的DH配置可以被国家级攻击者利用。

如果你不希望使用内置的标准DH参数组，最近的Apache版本还允许你配置自定义的DH参数。如果要做到这一点，请在`SSLCertificateFile`指令指定的文件末尾追加所需的参数。

13.12　TLS 会话管理

Apache支持两种会话管理机制：服务器端状态缓存和会话票证。Apache 2.2.x只有独立部署缓存的功能，但是Apache 2.4.x增加了分布式操作所必需的功能。

13.12.1　独立会话缓存

对于单台Web服务器，只有一个可用的TLS会话缓存选择：共享内存。也可以将会话缓存在DBF文件中，但这种方法在重负载下是不可靠的（根据Apache文档）。

---

① 在Apache 2.4.x上增加DHE参数长度，http://blog.ivanristic.com/2013/08/increasing-dhe-strength-on-apache.html（Ivan Ristić，2013年8月15日）。

对于使用共享内存缓存，你需要先启用mod_socache_shmcb模块。之后，在服务器范围内指定以下两条指令：

```
# 指定会话缓存类型、路径和大小 (1 MB)
SSLSessionCache shmcb:/path/to/logs/ssl_scache(1024000)

# 指定会话缓存最长持续时间为1天
SSLSessionCacheTimeout 86400
```

默认情况下，会将超时时间设置为5分钟，这是非常保守的。通常几乎没有理由去重新协商新的会话；我选择使用24小时代替默认值。默认的缓存大小为512 KB，但我增加到1 MB。对于规模较小的网站，这两个值可能能够较好地工作。知名网站都需要了解它们的使用模式，并且将缓存大小设置为适当的值。我用Apache 2.4.x测试，你应该使用1 MB缓存存储大约4000个会话。

> **注意**
> 重启动Apache（即使使用正常的选项保留主进程）会清除会话缓存。因此，每次重新启动会带来服务器的少量CPU开销以及对用户的响应延迟。一般情况下，你不应该担心它，除非重启非常频繁。

根据Apache的版本，对于TLS会话缓存，你可能还需要配置用于同步访问缓存的互斥锁。Apache 2.4.x默认使用互斥锁，但可以用Mutex指令调整配置。令人费解的是， Apache 2.2.x默认情况下不使用互斥锁，这意味着它的缓存在高负荷下很容易损坏。

要配置基于Apache 2.2.x的互斥锁，使用SSLMutex指令：

```
# 配置同步访问TLS会话缓存的互斥锁
SSLMutex file:/var/run/apache2/ssl_mutex
```

在Unix平台上，可靠的自动互斥锁选择历来很困难，因为一般不可能找到一个在所有系统上运行良好的互斥锁类型。出于这个原因，你会发现默认方案往往使用基于文件的互斥锁；它们是最可靠的，但不是最快的。

> **注意**
> Apache为整个服务器使用相同的TLS会话缓存，但不相关的应用程序共享会话缓存可能是危险的。会话恢复使用缩短的TLS握手，跳过证书验证。网络攻击者可以将流量从一个端口重定向到另一个可能绕过证书验证的端口，强制请求被不正确的程序处理。此攻击可能导致信息泄露。对于可能出现的问题的完整讨论参阅6.8节。

13

13.12.2 独立会话票证

默认情况下，由OpenSSL提供会话票证的实现。对于独立服务器，这种做法“正好有效”，尽管你应该清楚以下这些方面。

- 会话票证使用128位AES加密保护。当Web服务器最初启动时产生一次性密钥。取决于配置，有可能会使用多个密钥。
- 密钥长度是固定的，但128位对大多数用例来说已经足够安全。

- 重新启动服务器时，会产生新的票证密钥。这意味着，在重新启动后的所有连接都需要协商新的TLS会话。
- 相同的AES密钥对于服务器一直有效。为了尽量减少会话票证对前向保密的影响，应该确保定期重新启动Web服务器，最好是每天。

13.12.3　分布式会话缓存

如果你的网站是在多个服务器上运行，但你不能集中终止TLS（例如负载均衡设备），并且没有使用粘性会话（客户端始终发送到同一个节点），你将需要分布式TLS会话缓存，这是一种在集群节点之间交换会话信息的机制。

Apache 2.4.x使用流行网络缓存程序memcached，它对分布式TLS会话缓存的支持很好。要使用它，需要部署memcached缓存的一个实例，然后让所有Web服务器连接到它。

首先，确保你已经安装并且激活了mod_socache_memcache模块：

```
LoadModule socache_memcache_module modules/mod_socache_memcache.so
```

然后像这样配置TLS会话缓存：

```
# 为TLS会话缓存使用memcached
SSLSessionCache memcache:memcache.example.com:11211

# 指定缓存最长持续时间为1小时
SSLSessionCacheTimeout 3600
```

关于memcached的大小，请考虑以下要点。

- 独立服务器，分配足够的RAM，保证了会话数据可以缓存会话的整个持续时间（-m参数）。
- 锁定缓存（-k选项）以提高性能，并且防止敏感TLS会话数据写入交换分区（swap）。
- 确保允许连接的最大数足以涵盖整个集群（-c选项）支持的并发连接的最大数。

可以使用下面的配置文件作为起点定制：

```
# 以守护进程运行
-d

# 以用户名memcache运行
-u memcache

# 运行端口11211
-p 11211

# 日志文件
logfile /var/log/memcached.log

# 分配10 MB缓存
-m 10

# 允许连接数上限10240
-c 10240

# 锁定内存以提升性能，更重要的是阻止
# 敏感TLS会话数据被写入交换分区
-k
```

运行分布式TLS会话缓存似乎简单明了。在实践中，取决于具体细节，你需要考虑许多额外的问题，包括以下内容。

- 可用性

  Web服务器节点不再在本地保留任何TLS会话信息，而是依赖配置的memcached提供的数据。这意味着memcache是你集群的一个风险点。你将如何处理memcache运行不符合预期？

- 性能

  TLS会话数据在远程托管，memcache查找恢复的TLS连接会增加延迟。如果网络很快并且可靠，这个成本将是固定的，并且可能很小。衡量成本的唯一可靠方式是通过对比整个集群和独立服务器的性能。确保禁用客户端会话票证；否则你有可能测量了错误的恢复机制。

- 安全性

  与memcache的通信是不加密的，这意味着在内部网络中传输的敏感TLS会话数据有可能会暴露。这是不理想的，因为同一网络上的任何服务器的沦陷也会导致所有TLS会话的沦陷。这一问题可以通过使用某种特定的加密网络段与memcached通信来解决。

> **注意**
>
> 由于TLS会话缓存共享可能导致安全漏洞，最佳实践是永远不与无关的应用程序共享缓存。对于分布式缓存更是如此，很可能多个应用程序在服务器上使用同一缓存。为了获得最佳的安全性，应该为每个应用程序运行单独的memcache节点。

13.12.4　分布式会话票证

如果你部署Web服务器集群，并且期望每个节点能终止TLS，那么会话票证会引入额外的管理挑战。为了可靠地解密会话数据，所有集群节点必须使用相同的密钥；这意味着你可以不再依赖OpenSSL生成的每个服务器自己的密钥。

Apache 2.2.x不支持可配置的票证密钥。这意味着，如13.12.3节中所述，你唯一的选择是禁用会话票证。Apache 2.4.x支持通过SSLSessionTicketKeyFile指令手动配置会话票证密钥。有了它，你可以手动生成会话票证密钥文件，使用管理其他配置数据那样的机制，将其推送到集群中的所有节点。

会话票证密钥文件由48字节加密的随机数据组成。该数据被用于3个16字节（128位）的片段，分别用于密钥名称、HMAC密钥和AES密钥。

使用OpenSSL可以生成一个像这样的票证密钥文件：

```
$ openssl rand -out ticket.key 48
```

之后，你只需要给Apache指定密钥文件：

```
SSLSessionTicketKeyFile /path/to/ticket.key
```

13

> **警告**
> 会话票证密钥文件必须用与其他私有密钥一样的方式进行保护。虽然没有必要对其进行备份，但必须确保只有root用户可以访问该文件。此外，不同的应用程序始终使用不同的的会话票证密钥。这将确保一个站点的会话不会被另一个站点恢复。

对于独立部署的服务器，必须定期轮转会话票证密钥以尽量减少对前向保密的影响，比如每天一次。

13.12.5　禁用会话票证

尽管会话票证加入TLS协议是受欢迎的，但有时候你可能希望禁用它，通常是因为它引入额外的运行开销。从Apache 2.4.11开始，你可以使用SSLSessionTickets指令禁用会话票证。如果你运行的是旧版本并且不想升级，唯一的解决办法是给Apache源代码打补丁，下面是详细的说明。

要禁用Apache 2.2.x（对v2.2.27测试）的会话票证，使用下面的补丁：

```
--- ./modules/ssl/ssl_engine_init.c.orig    2014-07-16 10:53:06.000000000 +0100
+++ ./modules/ssl/ssl_engine_init.c    2014-07-16 10:53:44.000000000 +0100
@@ -615,6 +615,11 @@
      */
      SSL_CTX_set_options(ctx, SSL_OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION);
 #endif
+
+#ifdef SSL_OP_NO_TICKET
+    /* 禁用会话票证 */
+    SSL_CTX_set_options(ctx, SSL_OP_NO_TICKET);
+#endif
}
```

要禁用Apache 2.4.x的会话票证（测试过v2.4.10），使用下面的补丁：

```
--- ./modules/ssl/ssl_engine_init.c.orig 2014-07-14 05:29:22.000000000 -0700
+++ ./modules/ssl/ssl_engine_init.c 2014-07-21 08:07:17.584482127 -0700
@@ -583,6 +583,11 @@
         SSL_CTX_set_mode(ctx, SSL_MODE_RELEASE_BUFFERS);
 #endif

+#ifdef SSL_OP_NO_TICKET
+ /* 禁用会话票证 */
+ SSL_CTX_set_options(ctx, SSL_OP_NO_TICKET);
+#endif
+
     return APR_SUCCESS;
}
```

要禁用会话票证，首先要对源代码应用正确的补丁。可以通过定位包含源代码和补丁目录，并执行以下命令：

```
$ patch -p0 < disable-tickets-2.4.10.patch
patching file ./modules/ssl/ssl_engine_init.c
```

这里你需要按照自己的期望重新配置Apache并启用SSL_OP_NO_TICKET编译器标志。例如：

```
$ ./configure \
    --prefix=/opt/httpd \
    --with-included-apr \
    --enable-ssl \
    CFLAGS=-DSSL_OP_NO_TICKET
```

13.13 客户端身份验证

采用客户端身份验证的配置很简单：启用它，提供所有必要的CA证书，形成一个完整的验证链条，并提供吊销信息：

```
# 启用客户端身份验证
SSLVerifyClient require

# 指定从客户端证书到可信根证书路径的最大深度
SSLVerifyDepth 2

# 允许签发客户端证书的CA
# 这些证书的可分辨名称将被发送到每个用户，
# 以协助客户端证书选择
SSLCACertificateFile conf/trusted-certificates.pem
```

检查吊销客户端证书的传统方法是使用本地CRL列表。这种方法的性能最好，因为所有操作都在本地进。通常配置一个脚本并周期性地运转，用来检索新CRL并重新加载Web服务器：

```
# 启用客户端证书吊销检查
SSLCARevocationCheck chain

# 已吊销证书列表。该列表每次变动时都要求重新加载
# every time this list is changed.
SSLCARevocationFile conf/revoked-certificates.crl
```

从Apache 2.4.x开始，你也可以使用OCSP吊销检查。此选项以性能降低为代价提供实时吊销信息：

```
# 对客户端证书使用OCSP检查吊销
SSLOCSPEnable On
```

如果要求客户端身份验证，但客户端不提供的话，mod_ssl将以致命警报拒绝TLS握手。对于最终用户来说，这意味着他们得到一个隐藏的错误消息。通过SSLVerifyClient指令配置不同的值更加妥善地处理这种情况是可行的。

13

- optional

  TLS握手期间请求客户端证书，但并不是必需的。验证的状态存储在SSL_CLIENT_VERIFY变量中：NONE表示没有证书，SUCCESS表示有效的证书，FAILED后面带着验证失败的证书错误消息。如果你希望给那些证书验证失败的客户端提供自定义响应，这个功能非常有用。
- optional_no_ca

  TLS握手期间请求客户端证书，但不尝试验证。相反，它的期望外部服务会验证证书（SSL_CLIENT_family变量可用）。

> **注意**
> 使用可选的客户端验证可能会产生问题，因为有些浏览器不提示用户，或者按照此选项配置的其他方式选择客户端证书。其他一些浏览器对不能提供证书的网站不进行处理，这也会有问题。在你慎重考虑部署可选的客户端身份验证之前，用你的环境需要的所有浏览器做好测试验证。

由于性能原因，默认情况下mod_ssl不导出它的变量。如果你需要的话，通过使用SSLOptions指令配置所需的变量来启用导出：

```
# 向环境导出标准的mod_ssl变量和证书数据
SSLOptions +StdEnvVars +ExportCertData
```

13.14　缓解协议问题

Apache开发人员一般很快解决TLS协议相关的问题。在实践中，由于大多数部署都是基于各种操作系统发行的Apache版本，软件包的安全性取决于供应商。

13.14.1　不安全的重新协商

不安全的重新协商是2009年发现的协议漏洞，到2010年期间很大程度上得到了缓解。在这个问题被发现之前，Apache 2.2.x一直支持客户端发起的重新协商。2010年3月发布的版本2.2.15，不仅禁用了客户端发起的重新协商，而且对安全重新协商（RFC5746）也提供了支持。Apache 2.4.x最早于2012年初发布，这意味着它没有这个漏洞。

> **警告**
> 如果服务器发起的重新协商被使用，并且可以接受不支持RFC 5746的客户端，那么禁用客户端发起的重新协商不能完全解决此漏洞。这是因为攻击者可以连接到服务器，提交请求并启动服务器发起的重新协商，然后攻击受害者（客户端）。为了获得最佳的安全性，请检查SSL_SECURE_RENEG变量，以确保客户端支持安全的重新协商。

13.14.2　BEAST

从技术上说，可预测的IV漏洞（其另一个名称“BEAST攻击”更为人所熟知）存在于TLS 1.0以及更早的协议中，它同时影响通信的客户端和服务器端。在实践中，浏览器是脆弱的，因为利用漏洞需要攻击者能够控制发送（随后加密）什么数据。出于这个原因，BEAST不能通过服务器端补丁解决。

13.14.3　CRIME

2012年CRIME攻击利用了TLS协议层面的压缩。这个问题在协议层面一直没有解决，这就是为什么大家都禁用压缩。与CRIME攻击无关，Apache在2.2.24版本（2013年2月）和2.4.3版本（2012

年8月）中添加了SSLCompression指令，但保留默认启用压缩。[①]压缩在2.2.26版本（2013年11月）和2.4.4版本（2013年2月）中默认被禁用。

当谈到特定发行版的Apache，好消息是大部分厂商迄今为止已提供了安全补丁。例如，Debian在2012年11月[②]修复了自己的Apache版本，Ubuntu是在2013年7月[③]修复的。在Red Hat及其衍生分发版本中有一段时间需要通过操作环境变量来禁用压缩，[④]但Red Hat最终在2013年3月默认禁用了压缩。[⑤]

如果你的Apache版本支持TLS压缩，最好明确禁用它：

```
SSLCompression off
```

> **警告**
> 禁用压缩取决于OpenSSL 1.0.0或更高版本（SSL_OP_NO_COMPRESSION配置选项）提供的功能。较早版本的OpenSSL可能无法真正禁用压缩。

13.15　部署 HTTP 严格传输安全

由于HTTP严格传输安全（HTTP strict transport security，HSTS）是通过响应头进行激活的，在网站上配置它通常很容易。不过你也有可能会踩到某些陷阱，这就是为什么我建议你在作任何决定之前阅读10.1节。

HSTS用Header指令启用。最好总是保证该响应头对所有响应设置，包括错误：

```
# 启用HTTP严格传输安全
Header always set Strict-Transport-Security "max-age=31536000; includeSubDomains"
```

按照RFC，HSTS策略只能在加密通道的HTTP响应中进行设置。80端口上的同一站点不需要配置任何HSTS，但为了得到最好的效果，需要重定向到443端口。这将确保网站的所有访问者一访问网站就切换到HTTPS：

```
<VirtualHost *:80>
    ServerName www.example.com
    ServerAlias example.com
    ...
    # 将所有访问者重定向到加密的网站
    RedirectPermanent / https://www.example.com/
</VirtualHost>
```


① Bug #53219: mod_ssl should allow to disable ssl compression，https://bz.apache.org/bugzilla/show_bug.cgi?id=53219（ASF Bugzilla，2013年3月3日关闭）。

② DSA-2579-1 apache 2 — Multiple issues，https://www.debian.org/security/2012/dsa-2579（Debian，2012年11月30日）。

③ USN-1898-1: OpenSSL vulnerability，http://www.ubuntu.com/usn/usn-1898-1/（Ubuntu安全公告，2013年7月3日）。

④ Bug #857051: SSL/TLS CRIME attack against HTTPS, comment #5，https://bugzilla.redhat.com/show_bug.cgi?id=857051#c5（Red Hat Bugzilla，2013年4月19日关闭）。

⑤ RHSA-2013:0587-1，https://rhn.redhat.com/errata/RHSA-2013-0587.html（Red Hat，2013年3月4日）。

13.16　监视会话缓存状态

Apache通过mod_status模块暴露TLS会话缓存的状态，这是一个鲜为人知的事情。要启用此功能，首先启用附加的状态信息记录（在主配置上下文中）：

```
# 请求跟踪扩展状态信息
# 该指令只有Apache 2.2.x需要
# Apache 2.4.x在mod_status加载的时候会自动启用
ExtendedStatus On
```

在所需的位置配置mod_status的输出：

```
<Location /status>
    SetHandler server-status

    # 限制访问源IP
    # 不希望全世界都能看到敏感的状态信息
    Require ip 192.168.0.1
</Location>
```

> **警告**
> mod_status的输出包含敏感数据，这就是为何你必须始终限制访问。最好的办法是通过HTTP基本身份验证，但你又要多记住另一个密码。像在我的例子中的网络范围限制一样有用。

当你打开状态页面，在底部会看到类似这样的输出（我加粗了重点部分）：

```
cache type: SHMCB, shared memory: 512000 bytes, current entries: 781
subcaches: 32, indexes per subcache: 88
time left on oldest entries' objects: avg: 486 seconds, (range: 0...2505)
index usage: 27%, cache usage: 33%
total entries stored since starting: 12623
total entries replaced since starting: 0
total entries expired since starting: 11688
total (pre-expiry) entries scrolled out of the cache: 148
total retrieves since starting: 6579 hit, 3353 miss
total removes since starting: 0 hit, 0 miss
```

13.17　记录协商的 TLS 参数

Web服务器日志机制默认只关心HTTP请求和错误，不会告诉你很多关于TLS的使用。你要盯紧TLS运行，这主要有以下两个原因。

- 性能

  不正确的TLS会话恢复配置可能导致巨大的性能损失，这就是你为何要关注会话恢复的命中率。因此配置日志文件是非常有用的，可以确保服务器TLS会话恢复，协助缓存调优。只有Apache 2.4.x允许你通过SSL_SESSION_RESUMED环境变量实施。

- 协议和密码套件使用

  当要禁用安全性弱的协议版本和密码套件时，了解网站的用户群实际使用的协议版本和密码套件是很重要的。例如，SSL 2过去多年仍广受支持，因为人们害怕将其禁用。我们现在正面临着类似的问题：SSL 3协议、RC4和3DES加密。

假如你使用的是Apache 2.4.x，请使用以下指令来监控TLS连接：

```
# 使TLS变量对日志记录模块可用
SSLOptions +StdEnvVars

# 将每个请求的TLS信息记录到单独的日志文件
CustomLog /path/to/ssl.log "%t %h %k %X %{SSL_PROTOCOL}e\
 %{SSL_CIPHER}e %{SSL_SESSION_ID}e %{SSL_SESSION_RESUMED}e"
```

请注意以下几点：

- 当一个会话恢复后才会记录会话ID，并不会在初次请求中记录。
- SSL_SESSION_RESUMED变量：新的会话为Initial，恢复的会话为Resumed。
- %k变量记录在同一连接中的请求数。如果你在日志条目中看到一个零，就知道是它的第一个请求，正是被记录的这一次。
- %X变量记录请求结束时的连接状态。破折号意味着连接将被关闭，而加号意味着连接将保持打开状态。

Apache的日志记录功能与我们跟踪TLS处理的需求之间在细节方面有轻微的不匹配。TLS连接参数一般在连接的开始设置，期间不会改变，除非发生重新协商。ApacheCustomLog指令处理请求，这意味着对于许多HTTP事务的长连接，你会得到多个几乎相同的日志条目。%K变量在跟踪这一点方面很有用。一方面，这将使日志快速增长。另一方面，记录每一笔事务可以帮助你确定连接的恢复率，这对HTTP和TLS操作都是最有效的方式。

> **注意**
> 目前还没有办法记录TLS握手成功，但没有任何请求的连接。同样，也不可能记录TLS握手失败。

13.18 使用 mod_sslhaf 的高级日志记录

13

Apache的日志记录功能允许你确定连接使用的TLS参数，但不提供除此之外的任何信息。例如，你不知道每个客户端提供的最高协议版本和密码套件。如果有了这些信息，就可以确定用户的能力并实现最佳TLS配置，而无需经过试错的痛苦过程。

为了回答这些类似的问题，我编写了一个叫mod_ssl的Apache模块。此模块并不是深入Apache的钩子；相反，它以被动方式观察并解析所有TLS连接来提取客户端的能力。它可以提供以下有趣的信息。

- 支持的最高协议版本
- 已提供的密码套件列表

- 已使用的TLS扩展列表，特别是以下几项
  - SNI扩展的可用性
  - 对会话票证的支持
  - 对OCSP stapling的支持

除上述外，mod_ssl也可以记录整个原始的ClientHello，这对进行自定义握手的分析是非常有用的。还有一个名为SSLHAF_LOG的特殊变量，它只对一个连接上的第一个请求设置。这个变量旨在与Apache的条件日志记录功能一起工作，可以让你记录每个连接只需一条日志记录（从而节省了大量的磁盘空间）。

安装mod_sslhaf很简单。它没有正式的发行版，所以必须使用git来克隆源代码存储库：

```
$ git clone https://github.com/ssllabs/sslhaf.git
```

因为模块很小（大约只有1000行代码），文档包含在源代码本身之中，在文件mod_sslhaf.c中。要编译模块，执行下面的命令：

```
$ apxs -cia mod_sslhaf.c
```

命令行开关c、i、a代表编译（compile）、安装（install）和激活（activate）。根据你的配置文件，激活有时可能会失败。在这种情况下，模块通过添加以下行来配置手动激活（当然要使用系统的正确路径）：

```
LoadModule sslhaf_module /path/to/modules/mod_sslhaf.so
```

下面的配置使用所有mod_sslhaf功能并记录最重要的数据点，但每个连接只有一次：

```
# 使TLS变量对日志记录模块可用
SSLOptions +StdEnvVars

# 将每个请求的TLS信息记录到单独的日志文件
CustomLog /path/to/ssl.log "%t %h %k %X %{SSL_PROTOCOL}e\
 %{SSL_CIPHER}e %{SSL_SESSION_ID}e %{SSL_SESSION_RESUMED}e |\
 %{SSLHAF_HANDSHAKE}e %{SSLHAF_PROTOCOL}e %{SSLHAF_SUITES}e\
 %{SSLHAF_EXTENSIONS_LEN}e %{SSLHAF_EXTENSIONS}e \"%{User-Agent}i\""\
env=SSLHAF_LOG
```

此日志格式与13.17节中所使用的格式是相同的；在字符之后附加mod_sslhaf提供的信息。

> **提示**
>
> 大多数人永远不会考虑分析原始的ClientHello记录，这就是我没有在日志格式中包含它的原因。毕竟，它会占用很多空间，并且会影响日志记录的性能。如果你想跟踪此数据，可以查看它所在的变量SSLHAF_RAW。

第 14 章

配置Java和Tomcat

14

本章重点介绍了Java平台的TLS功能，包括不同版本功能的演进，但主要集中在Java 7和Java 8。我会从平台支持的加密特性开始讨论，然后介绍客户端和服务器相关的部署和配置，最后再来讨论非常流行的Java Web服务器Tomcat。

14.1 Java 加密组件

在Java中，需要以下多个组件的协同工作才能提供完整的SSL和TLS协议和相关功能。

- Java Cryptography Architecture（JCA）
  JCA提供了所有和加解密相关的统一架构。概念上讲，JCA只是一组抽象的API，并没有实际的代码。JCA的核心思想是，允许任意数量的Provider，针对API来实现具体的功能。
- Java Certification Path API
  Java Certification Path API（在Java参考文档中通常称为CertPath）负责处理证书以及证书路径相关的所有功能。就SSL/TLS而言，CertPath提供了处理X.509证书路径的API并符合PKIX标准，大多数SSL/TLS部署都依赖PKIX来建立信任。
- Java Secure Socket Extension（JSSE）
  JSSE是处理SSL和TLS协议的组件，基于JCA提供的加密算法和其他相关API。JSSE也是以一组API提供服务，允许不同的实现（可替换性）。
- JCA Provider
  Java自带了一些Provider，已经实现了各种加密算法，要安装新的Provider也很容易。默认配置对大多数情况已经足够使用，如果需要启用某个特殊功能或者性能调优，也可以明确指定需要使用和配置的Provider。
- `keytool`
  Java不是将密钥和证书存储在独立的文件中，而是将它们打包放在一个存储单元中，这种存储单元被称为密钥库（keystore）。要管理密钥库中的内容，你需要使用JDK提供的`keytool`工具。
- Java Root Certificates Store
  作为一个TLS库，如果没有包含可信证书，也就是常说的根（root）或者根证书（root certificate），那在公共互联网上根本用不起来。根证书的集合也被称为可信证书库（trust

store），JVM厂商通常维护着自己的可信证书库并随产品一起发布。[①]

在本节中，我的目标是为你提供所有与SSL/TLS相关的信息。如果你需要更深入的了解，建议直接查询Java 7[②]和Java 8[③]的参考文档。

14.1.1　无限制的强加密

Java加密强度可以有两种运行模式，两个模式下的代码是完全相同的，但通过配置加上了不同限制。默认情况下，所有安装都是在*强加密模式*（strong mode）下进行的，但是有一定限制，以符合美国密码学出口限制。在这种模式下，AES密码被限制为128位。另外一种模式称为*无限制加密强度*（unlimited strength），没有任何人为的限制。默认模式对大多数情况都已经够用，但是是仍然推荐使用无限制的强加密模式，以避免在某些特殊场景下出现互操作性问题（我会在本章后续部分详细讨论这些问题）。

如果想启用无限制模式（例如需要实现一个SSL评估工具，这个模式就很有用，因为你需要使用尽量多的密码套件来做评估），你需要从Oracle网站上下载特殊策略文件[④]，并按照安装说明将它们部署在磁盘的指定路径下。

> **注意**
> *在有些系统中，可能安装和使用了多个Java包，确保你修改的是正确的那个或者修改所有部署。即使系统只安装了一个版本，JDK和JRE通常也在不同目录下，需要分别修改。*[⑤]

14.1.2　Provider 配置

Java安装包自带了众多的Provider，有些是通用的，有些是具体平台专用的。Oracle的SSL/TLS实现（SunJSSE）是通用Provider的良好样例，同样的一份代码适用于所有平台。与之相反，SunMSCAPI Provider则是Windows操作系统上的专用组件，封装了加密功能的接口。

除了极少数情况以外，一般你无需修改Provider的配置。如果你有特定的功能需求或者想尝试提高性能，例如在下列情况下。

---

① Including Certificate Authority Root Certificates in Java，http://www.oracle.com/technetwork/java/javase/javasecaroo-tcertsprogram-1876540.html（Oracle，检索于2014年7月1日）。

② Java SE 7 Security Documentation，http://docs.oracle.com/javase/7/docs/technotes/guides/security/（Oracle，检索于2014年7月2日）。

③ Java SE 8 Security Documentation，http://docs.oracle.com/javase/8/docs/technotes/guides/security/（Oracle，检索于2014年7月2日）。

④ JCE Unlimited Strength Jurisdiction Policy Files for Java 7 and Java 8（http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html，http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html），（Oracle，检索于2014年7月2日）。

⑤ Patch-in-Place and Static JRE Installation，http://docs.oracle.com/javase/7/docs/webnotes/install/windows/patch-in-place-and-static-jre-installation.html（Java Platform Standard Edition 7 Documentation，检索于2014年7月2日）。

- 性能调优

  Java内置的加密功能并不是天然更慢[①]，但在实践中Java可能也不是性能最好的平台。有一些证据表明，使用OpenSSL和NSS可以提高加密性能，其中一个案例中的Intel测试用例声称，Java在使用NSS库时性能有38%的提升。[②]

- FIPS模式

  Java支持FIPS，但是需要使用经过FIPS认证的外部Provider，Mozilla的NSS是其中之一。

Provider的可替换性在你遇到bug或实现限制时十分有用，理论上讲，你可以通过更换Provider来解决这些问题，但通常这么做你可能只是从一组bug和限制转换到另外一组bug和限制。

14.1.3 功能概述

Java的SSL/TLS实现历来比较保守，一些核心的协议功能实现都相对较晚（参见表14-1）。在这个意义上说，Java库和其他平台十分相似（除了Microsoft的平台以外）。例如，Java 7中已经支持客户端的虚拟安全主机功能，但是直到Java 8中才支持服务器端的这一功能。类似地，尽管TLS 1.2在Java 7中已经加入支持，但直到在Java 8中才被默认启用。

表14-1 JSSE中SSL/TLS的功能演进

	Java 5 (2004年5月–2009年10月)	Java 6 (2006年12月–2013年2月)	Java 7 (2011年7月–2015年4月)	Java 8 (2014年3月–)
椭圆曲线加密	否[a]	是[b]	是[c]	是
客户端SNI	–	–	是	是
服务器端SNI	–	–	–	是
TLS 1.1和1.2	–	–	是[d]	是
AEAD GCM套件	–	–	–	是
SHA256和SHA384套件	–	–	是	是
大于1024的DH算法（客户端）	–	–	–	是
大于768的DH算法（服务器）	–	–	–	是[e]
安全重新协商	u26+	u22+	是	是
BEAST修复（1/*n*–1拆分）	–	u29+	u1+	是
OCSP stapling	–	–	–	–
服务器端密码套件优先	–	–	–	是
禁用客户端重新协商	–	–	–	是
AES硬件加速	–	–	–	部分支持[f]
默认客户端握手格式	v2	v2	v3	v3

① 最佳性能通常需要手动的汇编和优化，Java平台包含了大量的本地库和汇编代码，其中有的被用于加密操作。例如，Java 8对Intel和AMD处理器增加了AES加速的汇编代码。

② Improved AES Crypto performance on Java with NSS using Intel® AES-NI Instructions，https://software.intel.com/en-us/articles/improved-advanced-encryption-standard-aes-crypto-performance-on-java-with-nss-using-intel（Intel whitepaper，2012年4月6日）。

14

a Java 5中，JCA提供了EC API但未实现。

b Java 6中，JSSE加入了EC套件支持，但JDK没有实现任何EC算法。唯一默认支持EC套件的平台是Solaris，提供了本地EC功能并通过PKCS#11集成到Java中。

c Java 7正式版由SunEC provider实现了EC算法，然而OpenJDK中并未加入这一组件，要支持EC，可以使用第三方库(例如BouncyCastle)，或者通过PKCS#11集成一个本地库。

d 客户 端模式下默认禁用，服务器端模式下默认启用。

e 默认仅支持1024位，但是可以增加到2048位。

f JEP 164: Leverage CPU Instructions for AES Cryptography，http://openjdk.java.net/jeps/164（OpenJDK网站）。

14.1.4　协议漏洞

最新的Java版本已经修复了所有已知的SSL/TLS漏洞。尽管Java安全补丁发布得很频繁，但大部分漏洞实际只影响客户端软件，因此服务器端安装包通常很长时间才需要更新一次。然而偶尔也有服务器端bug的修复，有时甚至是加密库的问题。例如2014年4月发布的补丁修复了JSSE中的一个严重缺陷。[①]

升级服务器Java的另一个原因是更新系统可信证书库，可能影响需要与外部通信的Web应用。

- 不安全的重新协商

  Oracle在2010年3月30日发布了一个禁用重新协商的临时补丁[②]，用于解决不安全重新协商问题，之后在2010年10月12日发布的Java 5u26和Java 6u22中，正式支持了安全重新协商功能。Java 7和更高版本在首发时就已经支持安全重新协商。

  与其他客户端软件类似，Java客户端会连接到不支持安全重新协商的服务器，这很危险，因为客户端无法检测到不安全重新协商的攻击行为，即使客户端本身已经实现了安全重新协商也无济于事。另外一种选择是只允许客户端连接到支持安全重新协商的服务器，代价就是你将无法与不支持安全重新协商的服务器建立连接。[③]

- BEAST

  为了解决BEAST攻击问题，Java从Java 6u29和Java 7u1开始已经实现了1/*n*–1数据包拆分的方法。

- CRIME

  CRIME攻击利用的是压缩中的信息泄露，Java从未支持过TLS上的压缩，也就意味着Java客户端不受CRIME的影响。Java Web应用可能会受到CRIME的TIME/BREACH变种的影响，这两个变种攻击的是HTTP响应体本身的压缩。

14.1.5　互操作性问题

Java在服务器模式时，你不太可能遇到互操作性问题，Java支持多种协议和密码套件，理论

① Easter Hack: Even More Critical Bugs in SSL/TLS Implementations，http://armoredbarista.blogspot.co.uk/2014/04/easter-hack-even-more-critical-bugs-in.html（Chris Meyer，2014年4月16日)。

② Transport Layer Security (TLS) Renegotiation Issue Readme，http://www.oracle.com/technetwork/java/javase/documentation/tlsreadme2-176330.html（Oracle，检索于2014年7月2日）。

③ 根据2014年7月SSL Pulse的分析结果，所有测试网站中大约有11.6%不支持安全重新协商。

上你可以与所有客户端通信。

对客户端来说情况就完全不同，其中以下几个潜在问题需要特别注意。

- 根证书缺失

  JRE发布自带的根证书，可以让Java客户端在遇到不认识的网站时通信成功。随着时间的变化，老的根证书会过期，新的根证书会加入。如果某个网站使用过了一个不在可信证书库中的根证书，你就无法连接到这个网站。如果你没有定期更新JRE，可信根证书库就会过时从而造成连接失败，而老的证书库有可能还包含已经失信的根证书。此外，某些情况下官方的可信证书库可能没有包含你希望信任的根证书，这时就需要手动将根证书添加到证书库中。

- 服务器只支持256位密码套件

  有极少数网站被配置成了只支持256位密码套件，如果JRE没有更新到无限制模式（即只支持128位AES），你的客户端就无法访问这部分网站。

- 大于1024位的DH参数

  Java 8之前的所有版本都只支持最大1024位的客户端Diffie-Hellman（DH）参数，尽管现在还只有少数服务器使用了更强的加密位数，1024位的DH参数仍然被视为不安全的，部署更强的加密位数是一个趋势。

- 小于1024位的RSA密钥

  从7u40开始，Java拒绝连接到还在使用小于1024位RSA密钥的服务器。可以通过修改`jdk.certpath.disabledAlgorithms`属性来绕过这个限制，但通常来说这不是一个好主意。

- RC4的使用

  从8u51开始，Java从默认的客户端和服务器配置中删除了RC4算法[①]，这一变化会导致与互联网上一小部分只支持RC4算法的服务器连接失败。如果确实有必要，可以通过修改配置来明确启用RC4，否则不推荐这么做。

- MD2算法的根证书

  同样从7u40开始，Java已不再接受MD2签名的证书。目前还有一小部分服务器在证书链中包含了MD2证书，从而导致TLS连接失败。尽管可以在代码中重载MD2的拒绝行为，但这只应该用作你最后的手段。

更严格的算法限制

Java证书链路中的默认算法限制，可以通过禁用所有不安全的算法和密钥大小来加以改进，从而提高安全性。建议为`jdk.certpath.disabledAlgorithms`安全属性使用以下设置。

```
MD2, MD5, RSA keySize < 2048, DSA keySize < 2048, EC keySize < 256
```

① JDK 8u51 Update Release Notes，http://www.oracle.com/technetwork/java/javase/8u51-relnotes-2587590.html（Oracle，2015年7月14日）。

这些限制不一定会影响你的可信根证书。为了达到最佳效果，应该检查所有的根证书并移除弱算法的证书（使用上述条件）。

14.1.6　属性配置调优

Java公开了大量系统和安全属性，可以用来调整默认的加密设置。在本节中，我选择了一组最有用的设置（参见表14-2），你可以在JSSE文档中找到完整的属性列表。[①]

表14-2　对于SSL/TLS和PKI调优最有用的系统和安全属性

作　　用	属性名称	说　　明
HttpsUrlConnection使用的默认客户端协议	https.protocols	以逗号分隔的指定协议列表，例如，TLS 1.1、TLSv1.2。从Java 8开始可以用jdk.tls.client.protocols来配置所有SunJSSE客户端
HttpsUrlConnection使用的默认密码套件	https.cipherSuites	以逗号分隔的指定密码套件列表，供HttpsUrlConnection使用
启用Server Name Indication（SNI）	jsse.enableSNIExtension	Java 7及以后版本已默认启用，无需禁用除非遇到不兼容问题
允许不安全重新协商	sun.security.ssl.allowUnsafeRenegotiation	默认禁用，谨慎使用
允许客户端不安全重新协商	sun.security.ssl.allowLegacyHelloMessages	默认启用已兼容未升级的TLS客户端，理想情况下应该禁用但可能引起连接问题
禁用算法套件	jdk.tls.disabledAlgorithms	一个方便的设置，用于禁用某些算法而无需更改应用源代码。安全属性
禁用证书算法	jdk.certpath.disabledAlgorithms	证书算法限制。此参数的文档在java. security文件中。安全属性
重建不完整的证书链	com.sun.security.enableAIAcaIssuers	启用时Java客户端在遇到不完整的证书链时会根据已有的AIA信息尝试补全，默认禁用
启用证书吊销检查	com.sun.net.ssl.checkRevocation	默认禁用，启用时需要同时启用CRL或者OCSP
启用OCSP吊销检查	ocsp.enable	启用时Java客户端会使用OCSP来检查证书吊销信息，默认禁用。安全属性
启用CRL吊销检查	com.sun.security.enableCRLDP	启用时Java客户端会使用CRL来检查证书吊销信息，默认禁用，如果OCSP同时被启用则优先使用OCSP。安全属性

表14-3列出了Java 8中新增加的一些属性。

表14-3　Java 8中新增的系统属性

作　　用	属性名称	说　　明
禁用客户端不安全重新协商	jdk.tls.rejectClientInitiatedRenegotiation	设置为true禁用客户端不安全重新协商，本书写作时尚无公开文档

① Customizable Items in JSSE，http://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html#InstallationAndCustomization（JSSE 8 Reference Guide，检索于2014年7月2日）。

（续）

作　用	属性名称	说　明
配置服务器Diffie-Hellman密钥强度	jdk.tls.ephemeralDHKeySize	未定义时使用1024位，设置为legacy时兼容Java 7的行为自动匹配密钥长度，也可以设置为1024到2048之间的一个固定值
默认SunJSSE客户端协议	jdk.tls.client.protocols	作用与https.protocols类似，但影响所有SunJSSE客户端，而不是仅限于HttpsUrlConnection
指定老旧TLS算法	jdk.tls.legacyAlgorithms	只有在无更好选择可用的情况下，该属性中列出的算法和套件才会协商。该属性默认包含NULL、出口、匿名、RC4和DES算法。安全属性，自8u51起可用

系统属性（system property）和安全属性（security property）很类似，但两者的配置方法不同。有两种方法可以用来设置系统属性，第一种就是通过JVM命令行的-D开关，例如：

```
$ java -Dhttps.protocols=TLSv1 myMainClass
```

或者可以在运行代码中直接使用System.setProperty()方法：

```
System.setProperty("https.protocols", "TLSv1");
```

安全属性则不同，一般是通过修改$JAVA_HOME/lib/security/java.security文件来配置的。如果想通过命令行重载某些设置，那么在满足以下两个条件的情况下是可行的。

(1) 主配置文件中的security.overridePropertiesFile设置为true（默认值）。

(2) 不能在命令行中指定一个单独的属性，相反，你需要创建一个属性文件包含所有需要重载的属性。

如果满足以上两个条件，你可以用以下命令来重载默认安全属性。

```
$ java -Djava.security.properties=/path/to/my/java.security-overrides
```

此外，还有一个未正式公开的特性可以用来指定另一个完整的安全配置文件（而不仅仅是重载部分默认属性），通过使用两个等号：

```
$ java -Djava.security.properties==/path/to/my/java.security
```

在运行时，你可以通过Security.setProperty方法来设置一个安全属性，例如要改进默认的加密强度策略，你可以这样设置：

```
Security.setProperty("jdk.certpath.disabledAlgorithms",
    "MD2, MD5, RSA keySize < 2048, DSA keySize < 2048, EC keySize < 256");
```


> **警告**
>
> 在运行时设置属性是不可靠的，有些类只会在启动时获取属性值并一直使用，因此不受运行时属性变化的影响。为了避免混乱，建议只通过命令行开关或修改配置文件的方法来设置属性。

14.1.7　常见错误消息

当有意外发生时，JSSE会引发异常，但错误消息使用的语言往往过于技术性并且不能提供足

够的信息来帮助解决问题。本节中会介绍常见的JSSE错误消息以及处理这些错误的选项。

1. 证书链问题

当Java客户端在连接到服务器后证书验证失败时，会引发以下异常：

```
javax.net.ssl.SSLHandshakeException: sun.security.validator.ValidatorException: PKIX path
    building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find
    valid certification path to requested target
```

关于这个问题的根本原因，基本上有以下几种情况。

- 未知证书颁发者

  服务器的证书是由一个Java客户端不认识的CA所签发，这可能是因为你的可信证书库配置已过时而没有更新到新加入的CA，也可能是因为服务器使用的是一个自选的CA（没有被公共网络所认可）。如果你确信该CA是可信的，就可以把它加入到你的可信证书库中来解决问题。除此之外，信任任何一个未知根证书都是不推荐的。一旦加入，这个CA就可以冒充世界上任意一个网站了。

- 不完整的证书链

  尽管我们花了很多时间来讨论服务器证书，但在实际环境中需要配置的是完整的证书链。如果一个服务器的证书链缺失，客户端就无法建立一个到可信根证书的路径，解决方法是确保在服务器上配置正确的证书链。

  有时候不完整的证书链可以通过证书中的*颁发机构信息访问*（authority information access，AIA）扩展信息来加以补全，其中包含了上级证书的下载URL。Java默认并不支持AIA扩展，启用这个特性需要将`com.sun.security.enableAIAcaIssuers`属性设置为`true`。

- 自签名证书

  有不少服务器是通过自签名证书来运行的。如果它们希望向公众开放服务，这种方式是不可行的；如果不打算向公众开放服务则问题不大，可以通过将该证书加入到信任列表白名单中来解决问题。

> **警告**
>
> 与很多互联网上找到的“解决方案”相反，你绝不应该在代码中禁用证书校验来解决自签名证书问题。如果这样做，你的程序在遭遇**中间人**攻击时就惨了。简单来说，就是所有人都可以用任意证书来冒充你需要访问的网站。

2. 服务器主机名不匹配

当使用TLS连接到Web服务器时，正常情况下URL中的主机名应该与证书中的主机名相匹配。如果出现不匹配的情况，就会引发以下异常：

```
javax.net.ssl.SSLHandshakeException: java.security.cert.CertificateException: No name matching
   beta.feistyduck.com found
```

解决方案很简单，安装正确的证书并补全缺少的主机名。

3. 客户端Diffie-Hellman限制

所有Java 8之前的版本，都只支持最大1024位的Diffie-Hellman（DH）参数。如果运行于这些版本的Java客户端，遇到了使用大于1024位（通常就是2048位）DH参数的服务器，就会得到以下异常：

```
javax.net.ssl.SSLException: java.lang.RuntimeException: Could not generate DH keypair
...
Caused by: java.lang.RuntimeException: Could not generate DH keypair
...
Caused by: java.security.InvalidAlgorithmParameterException: Prime size must be multiple of 64,
    and can only range from 512 to 1024 (inclusive)
```

如果你对访问异常的服务器有控制权，可以通过以下方法简单地解决问题。

- 在服务器上启用并优先使用ECDHE套件，Java 7的客户端支持这些套件可以正常使用。
- 作为最后的手段，你可以将DH参数降级到1024位，当然这也降级了所有DH套件的安全性。在实际部署中，如果你配置了ECDHE套件，那就只有一小部分老客户端才会匹配到DH算法。选择这种方案时，请避免使用常用的（系统默认的）DH参数，更多相关信息参阅6.5节。
- 另一个最后的手段是完全禁用DHE套件，如果你同时启用了ECDHE套件，大部分客户端就会使用它，剩余的一小部分将失去前向保密的能力。

如果你倾向于在客户端来修改，可以尝试使用Bouncy Castle项目开发的JCE组件来替换Oracle的官方JCE组件（DH参数限制所在）。[①]对于这个方案我持保留意见，虽然它有时可以工作，但新增的Provider也可能带来其他让人费解的异常情况。要使用这一方案，你还需要将Java 6客户端的握手格式调整为v3，因为默认的v2格式不支持ECDHE。

4. 服务器名称标识不匹配

只有极少数服务器不兼容*服务器名称标识*（server name indication，SNI）的扩展，而从Java 7开始客户端默认会启用SNI。最常见的情况是，支持SNI的服务器在SNI信息中无法匹配任何一个虚拟主机时，返回了一个TLS警告。尽管TLS警告不是致命的，理论上可以忽略，但Java客户端在这种情况下会直接终止连接。当你升级JVM并看到以下异常时，应该就是碰到了上述问题：

```
javax.net.ssl.SSLProtocolException: handshake alert: unrecognized_name
```

5. 严格安全重新协商失败

当JVM在严格安全重新协商模式下运行时，需要TLS握手的两端都必须实现安全重新协商。不是这样的话，就会得到以下异常：

14

```
javax.net.ssl.SSLHandshakeException: Failed to negotiate the use of secure renegotiation
```

除非明确启用了严格安全重新协商（将`sun.security.ssl.allowLegacyHelloMessages`设置为`false`），才不会有此异常。如果是Java客户端遇到这个问题，最佳方案是升级服务器；如果不可行，那么唯一的选择就是切换回默认（不安全）模式。

① Provider Installation，http://www.bouncycastle.org/wiki/display/JA1/Provider+Installation（Bouncy Castle，检索于2014年7月2日）。

6. 协议协商失败

SSL 3是一个较旧的、过时的协议版本，不应该使用。理论上所有互联网的服务器都支持至少TLS 1.0，你不会遇到互操作性问题。但也可能会遇到只支持SSL 3的服务器，如果你禁用了SSL 3，在与这些服务器通信时就会得到以下异常：

```
javax.net.ssl.SSLHandshakeException: Server chose SSLv3, but that protocol version is not enabled or
    not supported by the client.
```

要解决这个问题，可以升级服务器或者降级客户端。

反过来说，如果没有启用新协议，你也可能遇到不支持TLS 1.0及更早版本的服务器。这种情况同样少见，但如果遇到，就会得到以下消息：

```
javax.net.ssl.SSLException: Received fatal alert: protocol_version
```

7. 握手格式不兼容

Java 6和更早的版本默认使用了SSL 2的握手格式，但不是所有服务器都支持。如果遇到了不支持的服务器，就会得到以下消息：

```
javax.net.ssl.SSLHandshakeException: Remote host closed connection during handshake
```

你可以重新配置客户端使用SSL 3握手格式来解决问题，在14.1.8节中的“在客户端上使用强协议”部分中有详细说明。

14.1.8　保护 Java Web 应用

在本节中，我将讨论关于如何在Java客户端或者Web应用中安全使用加密的主题。相关内容并不复杂，但是正确的信息却很难在互联网的资料海洋中找到。请注意，在此不会讨论加密以外的任何东西。例如，Cookie安全和会话管理安全都是相当复杂的主题，有很多可以讲的，但是完全覆盖这些内容已经超出了本书的范围。

1. 强制加密

你可以写一个Web应用并且期望它是安全运行的（即部署在TLS下），但并不能确保实际如此。可能是因为操作失误或者配置错误，你的程序就以明文HTTP方式开放使用了。

我的建议是在代码中强制检查应用是否是被安全访问的，方法是通过使用Servlet容器提供的`HttpServletRequest`实例上的`isSecure()`方法。对于已有的无法修改源代码的程序，可以通过Servlet Filter来加入检查。

> **注意**
>
> 代码检查可以找出明显的配置错误，但它不是万能的。有些系统是在上层架构中终止TLS的（例如，负载均衡或代理），并通过Web服务器配置参数来告之后端应用是否使用了加密。

2. 保护Web应用Cookie

以下代码片段创建了一个Cookie并且同时设置了httpOnly和secure标志，再将它添加到响应中（通过Servlet容器提供的HttpServletResponse实例）：

```
Cookie cookie = new Cookie(cookieName, cookieValue);
cookie.setMaxAge(cookieLifeInDays * 24 * 3600);
cookie.setHttpOnly(true);
cookie.setSecure(true);
response.addCookie(cookie);
```

显然，如果你有一个没有正确使用Cookie的应用，就需要检查源代码来找到创建Cookie的地方并改成安全模式。如果不希望修改源代码（或者无法获取源代码），可以尝试写一个Servlet Filter[①]来拦截Cookie的创建并强制设置为安全模式。

3. 保护Web会话Cookie

Java应用程序几乎完全依赖于Servlet容器来管理会话。在实践中，这意味着要确保会话Cookie安全，需要修改配置。

对于符合Servlet 3.0或更高版本规范[②]的应用，这很容易实现，配置中已经支持保护会话Cookie的设置项。要启用保护，只需要在应用的web.xml文件中加入以下片段：[③]

```
<session-config>
    <cookie-config>
        <secure>true</secure>
        <http-only>true</http-only>
    </cookie-config>
</session-config>
```

对于使用较早Servlet规范版本的应用，其行为完全依赖容器，有些产品在加密启用时会自动以安全模式创建Cookie。

4. 部署HTTP严格传输安全

HTTP严格传输安全（HTTP strict transport security，HSTS）是一项新技术，可以让不期望明文交互的Web应用强制启用加密传输，在第10章中有详细介绍。要部署HSTS，需要在应用中增加一个响应头，只需要调用一个方法：

```
response.setHeader("Strict-Transport-Security",\
    "max-age=31536000; includeSubDomains; preload");
```

实际上，在Web服务器层面配置安全策略通常是更好的方式。Java应用也可以使用Servlet Filter，不需要自己写，可以从现有的开源项目（例如HeadLines项目[④]）中直接借用。

14

① The Essentials of Filters，http://www.oracle.com/technetwork/java/filters-137243.html（Oracle，检索于2014年7月2日）。

② JSR-000315 Java™ Servlet 3.0，http://download.oracle.com/otndocs/jcp/servlet-3.0-fr-eval-oth-JSpec/（Java Community Process，2009年12月）。

③ 可以在代码中设置当前ServletContext的SessionCookieConfig实例来达到同样的效果，最好在ServletContext-Listener创建ServletContext之后立刻设置。

④ HeadLines，https://github.com/sourceclear/headlines（SourceClear，检索于2014年7月1日）。

5. 在客户端上使用强协议

对于客户端应用，Java默认的协议配置历来是专注于互操作性而不是安全性。例如Java 6中还在使用SSL 2握手格式，其实这种格式只有在使用SSL 2协议时才是必要的，而Java实际从未支持过SSL 2。Java 7不再支持SSL 2握手格式，但仍然没有对客户端默认启用TLS 1.1和TLS 1.2，尽管它实际已经支持了这些新协议（对于服务器，默认是启用的）。Java 8则同时在客户端和服务器默认启用了TLS 1.1和TLS 1.2。

如果只使用了`HttpsUrlConnection`类，那最简单的方法就是修改`https.protocols`系统属性，请参考之前在14.1.6节中的讨论，通过设置`https.protocols`来修改这个类的默认协议配置。从Java 8开始，`jdk.tls.client.protocols`系统属性可以起到相同的作用，并且会对所有依赖SunJSSE的代码都生效。

如果是一个应用开发并且不能控制程序运行的环境，此时修改系统属性可能不适用。最好的办法是在程序中确保应用使用了指定的协议。如果可以直接操作底层套接字对象，这个工作很简单直接，你可以调用`SSLSocket.setSSLParameters()`来设置自定义的配置。

但是大多数情况下套接字过于底层，这也就是为什么你通常会发现自己使用的是更高级别的`HttpsURLConnection`类。不幸的是，修改这个类使用的协议要困难得多，你需要创建一个自定义的`SSLSocketFactory`并且确保一直使用这个自定义类。

以下是我的自定义工厂，启用了所有支持的协议（协议版本没有使用硬编码，与未来新协议兼容）但禁用了SSL 2握手格式和SSL 3协议：

```
import java.io.IOException;
import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.List;

import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;

public class MySSLSocketFactory extends SSLSocketFactory {

    private String enabledProtocols[] = null;

    private String enabledCipherSuites[];

    private SSLSocketFactory sslSocketFactory;

    public MySSLSocketFactory() {
        sslSocketFactory = (SSLSocketFactory) SSLSocketFactory.getDefault();
        enabledCipherSuites = sslSocketFactory.getDefaultCipherSuites();
    }

    private Socket reconfigureSocket(Socket socket) {
        SSLSocket sslSocket = (SSLSocket) socket;
```

```
    if (enabledProtocols != null) {
        sslSocket.setEnabledProtocols(enabledProtocols);
    } else {
        List<String> myProtocols = new ArrayList<String>();

        for (String p : sslSocket.getSupportedProtocols()) {
            if (p.equalsIgnoreCase("SSLv2Hello")
                    || (p.equalsIgnoreCase("SSLv3"))) {
                continue;
            }

            myProtocols.add(p);
        }

        sslSocket.setEnabledProtocols(myProtocols
                .toArray(new String[myProtocols.size()]));
    }

    sslSocket.setEnabledCipherSuites(enabledCipherSuites);

    return socket;
}

public void setEnabledProtocols(String[] newEnabledProtocols) {
    enabledProtocols = newEnabledProtocols;
}

public void setEnabledCipherSuites(String[] newEnabledCipherSuites) {
    enabledCipherSuites = newEnabledCipherSuites;
}

@Override
public Socket createSocket(Socket s, String host, int port,
        boolean autoClose) throws IOException {
    return reconfigureSocket(sslSocketFactory.createSocket(s, host, port,
            autoClose));
}

@Override
public String[] getDefaultCipherSuites() {
    return enabledCipherSuites;
}

@Override
public String[] getSupportedCipherSuites() {
    return sslSocketFactory.getSupportedCipherSuites();
}

@Override
public Socket createSocket(String host, int port) throws IOException,
        UnknownHostException {
    return reconfigureSocket(sslSocketFactory.createSocket(host, port));
}
```

```
    @Override
    public Socket createSocket(InetAddress host, int port) throws IOException {
        return reconfigureSocket(sslSocketFactory.createSocket(host, port));
    }

    @Override
    public Socket createSocket(String host, int port, InetAddress localHost,
               int localPort) throws IOException, UnknownHostException {
            return reconfigureSocket(sslSocketFactory.createSocket(host, port,
                    localHost, localPort));
        }

        @Override
        public Socket createSocket(InetAddress address, int port,
                InetAddress localAddress, int localPort) throws IOException {
            return reconfigureSocket(sslSocketFactory.createSocket(address, port,
                    localAddress, localPort));
        }
    }
```

之后，任何时候当你创建HttpsUrlConnection实例时，都需要设置和使用自定义的工厂：

```
URL u = new URL("https://www.feistyduck.com");
HttpsURLConnection uc = (HttpsURLConnection) u.openConnection();
uc.setSSLSocketFactory(new MySSLSocketFactory());
```

6. 证书吊销检查

默认情况下，Java对访问的证书不会执行任何证书吊销检查，这有潜在的不安全性。要达到最高的安全性，应该同时启用CRL和OCSP吊销检查，即将com.sun.net.ssl.checkRevocation、ocsp.enable和com.sun.security.enableCRLDP设置为true。

此外，你也应该允许Java尝试重建不完整的证书链，方法是通过设置com.sun.security.enableAIAcaIssuers属性，否则不完整的证书链无法通过验证，会导致与服务器的通信失败。

14.1.9　常见密钥库操作

本节将讨论关于密钥和证书管理最常见的一些工作，keytool工具可以帮助你完成大部分相关工作，但你可能还需要求助于使用OpenSSL来执行某些任务，特别是密钥和证书的导入。

> **注意**
> 如果不喜欢在命令行上花时间，可以考虑使用一个叫作KeyStore Explorer的工具[1]，对常见的keytool操作提供了一个友好的用户界面。

1. 密钥库布局

有一个不太明显的特性，Java实际上可以允许使用任意数量的密钥库。对于客户端来说，通常你不需要用到这个特性，因为使用系统提供的根密钥库已经足够了。如果你会定期更新JRE，系统

① KeyStore Explorer，http://keystore-explorer.sourceforge.net/（检索于2014年7月1日）。

密钥库也会自动保持更新，否则，你可能需要时不时地手动更新密钥库以加入新的可信根证书。

对于服务器而言则完全不同，使用多个密钥库不仅是可能的，而且是值得推荐的。除非有很好的理由不这么做，否则应该永远保持每个站点使用独立的密钥库。这种方案的优点是：(1) 网站密钥的安全性保持独立，可以使用不同的密码；(2) 将网站从一个服务器迁移到另一个服务器很容易。

在同一个密钥库内，每个证书链都必须有唯一的别名。如果采用了以上服务器密钥库的使用建议，则无需考虑关于别名的问题，因为在一个密钥库中只存在唯一的证书链。在本章的其余部分，我假设都是这种情况，并且一直会使用别名server。

2. 创建一个自签名证书和密钥

要创建一个自签名证书和私钥，请使用-genkeypair命令[①]：

```
$ keytool -genkeypair \
    -keystore feistyduck.jks \
    -alias server \
    -keyalg RSA \
    -keysize 3072 \
    -validity 365 \
    -ext SAN="DNS:www.feistyduck.com,DNS:feistyduck.com"
Enter keystore password: ****************
Re-enter new password: ****************
```

在本例中，我使用了keytool的一个允许创建多域名证书的特性（-ext开关），这个特性在Java 6和之前的版本中没有提供。

> **警告**
> keytool工具可以通过命令行的-storepass开关来传入密钥库密码，但是我不推荐要这样使用，如果这样做，密码会被记录在命令行的历史信息中，也可能会被其他人从进程列表中看到。

当你输入密码后，会提示你输入证书所需要的信息。第一个问题有点误导性，不是在询问你的姓名，而是要你输入所需的域名（例如，www.feistyduck.com）：

```
What is your first and last name?
  [Unknown]: www.feistyduck.com
What is the name of your organizational unit?
  [Unknown]: Engineering
What is the name of your organization?
  [Unknown]: Feisty Duck Limited
What is the name of your City or Locality?
  [Unknown]: London
What is the name of your State or Province?
  [Unknown]: England
What is the two-letter country code for this unit?
  [Unknown]: GB
Is CN=www.feistyduck.com, OU=Engineering, O=Feisty Duck Limited, L=London, ST=England, C=GB correct?
```

① Java 7之前，这个命令是-genkey。

```
  [no]: yes
Enter key password for <server> (RETURN if same as keystore password):
```

现在可以检查一下密钥库中的结果，用以下命令查看你的密钥和证书内容：

```
$ keytool -keystore feistyduck.jks -list -v
Enter keystore password: ****************
[...]
Alias name: server
Creation date: 01-Jul-2014
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=www.feistyduck.com, OU=Engineering, O=Feisty Duck Limited, L=London, ST=England, C=GB
Issuer: CN=www.feistyduck.com, OU=Engineering, O=Feisty Duck Limited, L=London, ST=England, C=GB
Serial number: 4f3326e0
Valid from: Tue Jul 01 17:10:31 BST 2014 until: Wed Jul 01 17:10:31 BST 2015
Certificate fingerprints:
         MD5: 55:63:0B:F5:F5:45:67:62:2D:85:FE:5C:D2:8E:1E:27
         SHA1: A4:AD:C6:1E:F6:1F:73:B0:BD:C6:2F:83:F5:B1:67:82:61:94:89:CE
         SHA256: FD:0A:BE:5B:9F:93:9D:BA:DF:FD:54:8B:37:0A:A4:7C:92:1F:03:25:8C:01:ED:92:9B:BE:
            AA:19:68:27:B9:4D
        Signature algorithm name: SHA256withRSA
        Version: 3

Extensions:

#1: ObjectId: 2.5.29.17 Criticality=false
SubjectAlternativeName [
  DNSName: www.feistyduck.com
    DNSName: feistyduck.com
]

#2: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 02 14 B4 49 F6 15 F0 77   FE 9A C8 86 2A 02 10 95  ...I...w....*...
0010: 9A 46 FD EB                                        .F..
]
]
```

3. 创建一个证书签名申请

在你创建完自签名证书和密钥后，还需要前进一小步来创建证书签名申请（certificate signing request，CSR）：

```
$ keytool -certreq \
    -keystore feistyduck.jks \
    -alias server \
    -file fd.csr
Enter keystore password: ****************
```

现在可以将文件fd.csr提交给你的CA用于签发正式证书了。

4. 证书导入

从CA接收到服务器证书后，你需要将它以及构建完整证书链需要的所有证书全部导入密钥

库。首先，导入根证书：

```
$ keytool -import \
    -keystore feistyduck.jks \
    -trustcacerts \
    -alias root \
    -file root.crt
```

然后，使用相同的命令（但每次使用不同别名），导入中间证书：

```
$ keytool -import \
    -keystore feistyduck.jks \
    -trustcacerts \
    -alias intermediate1 \
    -file intermediate1.crt
```

最后，导入服务器证书：

```
$ keytool -import \
    -keystore feistyduck.jks \
    -alias server \
    -file fd.crt
```

> **注意**
> keytool的一个特别棒的地方就是它会检查被导入的证书与密钥是否匹配，以及证书链是否有效。根据我的研究，大约有6%的服务器部署了错误的证书链，keytool的这个功能可以避免类似错误的发生。

5. 已有证书的格式转换

如果要从现有服务器（例如Apache）迁移，往往需要将一组密钥和证书文件合并到一个单独的密钥库中。Keytool工具无法完成这个任务，但用OpenSSL来做则很容易。

以下命令利用已有的密钥和证书，并把它们转换成一个新的pkcs12格式的密钥库中：

```
$ openssl pkcs12 -export \
    -out feistyduck.p12 \
    -inkey fd.key \
    -in fd.crt \
    -certfile fd-intermediates.crt \
    -name server
Enter Export Password: ****************
Verifying - Enter Export Password: ****************
```

如果你有多个中间证书，请把它们合并到一个文件中（参见例子中的fd-intermediates.crt文件）。

你可以直接使用这个新的密钥库，但因为它不是Java原生格式，你可能需要在配置中指定对应的格式。例如在Tomcat中，可以将keystoreType参数设置为pkcs12。

如果你喜欢保持一切不变，也可以用keytool把密钥库转换回原生的（JKS）格式：

```
$ keytool -importkeystore \
    -srckeystore feistyduck.p12 \
    -srcstoretype pkcs12 \
    -destkeystore feistyduck.jks
```

```
Enter destination keystore password: ****************
Re-enter new password: ****************
Enter source keystore password: ****************
Entry for alias server successfully imported.
Import command completed: 1 entries successfully imported, 0 entries failed or cancelled
```

6. 客户端根证书导入

有时可能会遇到这么一种情况：尽管服务器证书是由公共CA签发的，但你的Java客户端却无法连接到指定的服务器。在这种情况下，需要将缺失的根证书加入到你的密钥库中。

首先需要获取缺失的根证书，这很容易，因为现在所有的浏览器都提供了证书查看功能。直接访问指定网站，选择查看证书的功能，将根证书导出到一个文件中。注意这里不需要导出中间证书。

然后执行以下命令：

```
$ keytool -import \
    -keystore /path/to/keystore.jks \
    -trustcacerts \
    -file /path/to/root.crt \
    -alias UNIQUE_ROOT_ALIAS
```

> **注意**
>
> 如果想创建一个自定义密钥库给应用程序使用，你可以使用任意密码。如果只在密钥库中保存根证书，密码实际并不重要。如果想替换Java默认的密钥库，使用changeit作为密码，以与默认密码保持一致。

建议在一个独立的区域维护你的主密钥库，并按需分发。如果要修改默认的Java密钥库，可以简单地将你的密钥库复制到对应的路径，大部分情况下就是`$JAVA_HOME/jre/lib/security/cacerts`。

14.2　Tomcat

如果想在Java平台上运行Web服务，那么你很可能会依赖Tomcat或者它的某个衍生产品。在Tomcat中使用TLS有些混乱，因为可以有很多种不同的方法来实现：

- 不在Tomat上运行TLS

  从历史上看，有相当数量的Tomcat是部署在Apache反向代理之后的。Apache不仅流行而且功能强大，拥有各种各样的模块可以支持所有能想到的功能，这也使它可以形成一个单独的架构层用来处理所有与HTTP相关的功能，而让Tomcat专注于Java特有的功能。这种架构是如此流行，以至于Apache有一个单独的代理模块（`mod_proxy_ajp`[①]）使用自定义协议AJP来直接对接Tomcat。

① Module mod_proxy_ajp，https://httpd.apache.org/docs/trunk/mod/mod_proxy_ajp.html（Apache httpd trunk documentation，检索于2015年7月22日）。

在这种模式下，所有与TLS相关的功能都是在Apache上配置的。因此对那些已经有Apache使用经验或者希望规避Java和Tomcat的TLS限制的用户来说，这种方案很有吸引力。

- 使用JSSE

  如果想在Tomcat层面上终止TLS，默认的选择就是使用JSSE。这个方案简单直接，因为所有Java安装包无需任何调整都已经支持了。同样，这个方案也意味着接受了JSSE的所有限制。尽管如此，Java 8中所作的大量改进，已经完全可以让JSSE成为一个强大安全的服务器运行平台。

- 使用APR和OpenSSL

  为了进一步提高Tomcat性能，开发者想出了一个使用本地库的方案，也就是Tomcat Native[①]，这个库包装了其他两个成熟的本地库：APR（Apache Web服务核心）和OpenSSL。Tomcat在启动时如果发现有Tomcat Native库，就会自动加载它。有一些不成熟的证据表明使用Tomcat Native后性能有所提高，但由于这个库同样也接管了套接字处理和其他I/O操作，所以很难说明性能的提高是由于I/O还是因为OpenSSL。在启动的时候，Tomcat自己也会提示使用Tomcat Native后提高了性能。

  Tomcat Native的一个主要缺点是增加了部署的复杂性，它是一个需要独立安装和维护的组件。Tomcat Native和JDK版本有绑定，也就是说当升级Java版本时，需要重新编译Tomcat Native。

  对于Windows来说，Tomcat Native有二进制文件提供。有些平台，例如Ubuntu，直接把Tomcat Native作为一个可选包提供（在Ubuntu中包名是`libtcnative-1`），但是这个版本可能太老无法在最新版本的Tomcat上使用。此外，新的Tomcat Native版本包含了很多重要的改进。

  当你决定使用OpenSSL时，Java加密库的功能和性能就不再起作用，它完全依赖于你使用的Tomcat Native和OpenSSL以及这些组件本身支持的功能。

让事情更复杂的是，使用JSSE的Tomcat支持三种连接器（处理传入连接的服务器组件）：老的BIO（blocking，阻塞）、默认的NIO（nonblocking，非阻塞）以及最新的NIO2（also nonblocking，也非阻塞）。[②]如果你使用的是OpenSSL，只有一种混合使用阻塞和非阻塞的连接器。

表14-4（从Tomcat文档中复制而来）展示了不同选项的一个对比：

表14-4　不同Tomcat连接器的性能选项对比

	Java BIO	Java NIO	Java NIO2	Tomcat Native
类名称	`Http11Protocol`	`Http11NioProtocol`	`Http11Nio2Protocol`	`Http11AprProtocol`
Tomcat版本	3.x以上	6.x以上	8.x以上	5.5.x以上
支持轮询	否	是	是	是

① Tomcat Native，https://tomcat.apache.org/native-doc/（Apache Software Foundation，检索于2015年7月22日）。

② 阻塞/非阻塞的名称是用来说明TCP连接是如何被处理的。一个阻塞连接器会使用整个线程来服务于一个TCP客户端，一个非阻塞连接器可以在一个线程或者一个小线程池中处理所有TCP客户端。阻塞连接器适用于处理速度快的客户端，而非阻塞连接器更适用于大量慢的客户端。Tomcat 7默认使用BIO连接器，而Tomcat 8默认使用NIO连接器。

（续）

	Java BIO	Java NIO	Java NIO2	Tomcat Native
轮询队列大小	无	maxConnections	maxConnections	maxConnections
读HTTP请求头	阻塞	非阻塞	非阻塞	阻塞
读HTTP请求体	阻塞	模拟阻塞[a]	阻塞	阻塞
写HTTP响应	阻塞	模拟阻塞	阻塞	阻塞
请求等待	阻塞	非阻塞	非阻塞	非阻塞
SSL实现	Java（JSSE）	Java（JSSE）	Java（JSSE）	OpenSSL
SSL握手	阻塞	非阻塞	非阻塞	阻塞
最大连接数	maxConnections	maxConnections	maxConnections	maxConnections

a 尽管连接器是非阻塞的，但传统的Servlet规范对请求体的处理要求使用阻塞I/O，因此，非阻塞连接器在这里模拟了阻塞I/O。Servlet 3.1规范（Tomcat 8中支持）引入了非阻塞I/O。

不幸的是，很难找到好的文档来帮助我们决定在什么情况下使用哪种方案最好。如果你关注性能，有一些所谓的证据表明使用前置反向代理终止TLS是最快的方案，其次是Tomcat Native，最后是JSSE，而BIO/NIO/NIO2连接器的选择对性能并无太大影响，但是这些可能都依赖于实际的使用场景。不仅如此，性能也只是决策的一个方面，当考虑TLS时，实际能支持的功能可能更重要。表14-5总结了JSSE（Java 7和Java 8）、Tomcat Native以及前置Apache反向代理终止TLS的一些差别：

表14-5　不同TLS终止方式下TLS功能的对比

	Tomcat (Java7)	Tomcat (Java 8)	Tomcat Native	Apache 2.4.x
强DH参数；位数	否；768	底线；1024	底线；1024	是；2048+（2.4.7）
配置强DH参数	–	是	–	是（2.4.7）
椭圆曲线支持	是	是	是（1.1.30）	是
密码套件优先	–	是（7.0.61和8.0.21）	是	是
虚拟安全主机	–	尚未支持[a]	–	是
禁用客户端重新协商	–	是	是	是
TLS会话缓存控制	是	是	–	是
TLS集群会话缓存	–	–	–	是
会话票证支持	–	–	是	是
禁用会话票证	–	–	–	是（2.4.11）
会话票证配置	–	–	–	是
OCSP stapling	–	–	–	是
多密钥支持[b]	–	–	–	是

a 尽管Java 8的JSSE支持，但这一功能的使用需要Tomcat的代码改造，本书写作时尚未提供，预计在Tomcat 9中支持。
b 底层的JSSE引擎从Java 7后就支持多密钥功能了，但Tomcat中没有用到。

表14-5中列出的一些特性属于高级特性，只影响特定用户，但有些是很基本的特性，在Java 7和更早版本中却有重大限制。

- 不安全的DHE套件

在Java 8中，服务器的临时Diffie-Hellman（DH）套件默认使用1024安全位，有利于互操

作性但对安全性而言并不是最佳。可以通过设置`jdk.tls.ephemeralDHKeySize`系统属性将强度提高到2048位。

在Java7和更早的版本中，服务器的临时DH参数被限制为768位，因此在升级到Java 8之前，请不要在JSSE中使用任何临时DH套件。

- 密码套件优先

  在Java 8之前的版本中，JSSE不允许服务器控制密码套件顺序，这意味着客户端列表中的第一个支持的套件会被使用，在实践中这限制了强制使用安全配置的能力。例如，你无法在配置中支持RC4而只能将其用于那些没有更好方案的客户端，同样，你也不能强制优先选择支持前向保密的套件。

  从Java 8开始，JSSE已经支持服务器密码套件优先，但是服务器应用需要修改来启用这个新特性。Tomcat在7.0.61和8.0.21中已加入支持。[①]

- 禁用客户端重新协商

  客户端重新协商是协议的一个特性，但实际没有什么使用价值，反而是给攻击者提供了一个强制服务器持续握手来消耗CPU资源的DoS攻击的机会。这里的问题在于一次TCP连接原则上可以执行多次握手，而大多数DoS检测技术都是基于连接频率来实现的，导致持续握手攻击很难防御。

  从Java 8开始，你可以通过未公开的系统属性`jdk.tls.rejectClientInitiatedRenegotiation`来禁用客户端重新协商。

由于问题众多，在将Java Web服务器升级到支持服务器密码套件优先前，我建议你使用Tomcat Native（1.1.30以上）或者前置Apache反向代理来终止TLS。

14.2.1 TLS 配置

要配置TLS[②]，你需要设置Tomcat配置中`Connector`元素的一系列属性。`protocol`属性决定了需要使用三种连接器中的哪一种，使用默认值（HTTP/1.1）时Tomcat首先会尝试使用APR连接器。如果APR连接器不可用，Tomcat 7及更早版本会降级到BIO连接器，而Tomcat 8会使用NIO连接器。

你不应该在生产环境中依赖默认设置，而是应该在`protocol`属性中明确配置所需的连接器名称，如随后几节所述。

在JSSE中使用阻塞连接器（blocking connector，BIO）：

```
<Connector
    protocol = "org.apache.coyote.http11.Http11Protocol"
    port = "443"
    ...
/<
```

① Bug #55988: Add parameter useCipherSuitesOrder to JSSE(BIO and NIO) connectors，https://bz.apache.org/bugzilla/show_bug.cgi?id=55988（ASF Bugzilla，检索于2015年3月27日）。

② SSL Support，https://tomcat.apache.org/tomcat-8.0-doc/config/http.html#SSL_Support（Apache Tomcat 8 Documentation，检索于2014年7月2日）。

在JSSE中使用非阻塞连接器（nonblocking connector，NIO）：

```
<Connector
    protocol = "org.apache.coyote.http11.Http11NioProtocol"
    port = "443"
    ...
/<
```

默认情况下，Tomcat会寻找Tomcat Native并启用，这是在`AprLifecycyleListener`类中实现的，其参数在后面会介绍到。如果你不希望使用Tomcat Native，可以直接禁用这个类，如果只是希望禁用OpenSSL（保留APR），可以将`SSLEngine`参数设置为`off`：

```
<Listener
    className = "org.apache.catalina.core.AprLifecycleListener"
    SSLEngine = "off"
```

反过来说，如果你保留了Tomcat Native并希望使用OpenSSL，可以把`protocol`属性设置为`Http11AprProtocol`类：

```
<Connector
    protocol = "org.apache.coyote.http11.Http11AprProtocol"
    port = "443"
    ...
/<
```

外部TLS终止

即使你在外部做TLS终止，有些TLS配置在Tomcat上也是必要的。在外部TLS终止时，虽然整个部署架构是安全的，但Tomcat对TLS无感知，在Tomcat中运行的应用也同样无感知，这会导致一些细节性和安全性问题，例如，会话Cookie没有被标记上安全属性，导致会话可能被外部劫持。

如果你是在Apache中使用`mod_jk`或`mod_proxy_ajp`向后连接到Tomcat，两者都实现了AJP通信协议，你无需做任何额外工作，因为AJP协议会把TLS信息透传到Tomcat。

在其他情况下，你需要在配置和信息交互上做更多的工作，例如要让Tomcat知道TLS已在外部终止，需要配置`scheme`和`secure`字段：

```
<Connector
    scheme = "https"
    secure = "true"
    ...
>
```

至于信息交互，你可以使用Tomcat的SSL Valve功能[1]，可以从请求头中提取信息（由终止TLS的代理设置），并将信息输出到对应的Tomcat数据结构中。

如果以上解决方案对你都无效，那就需要编写一个自定义扩展来继承你的代理，将安全标识、

① SSL Valve，http://tomcat.apache.org/tomcat-8.0-doc/config/valve.html#SSL_Valve（Tomcat 8 documentation，检索于2014年6月26日）。

正确的访问端口、协议格式等传到Tomcat。[①]

14.2.2 JSSE 配置

以下配置片段在443端口启用TLS并且显式配置了所有参数（除了客户端证书身份验证之外，极少使用）：

```
<Connector
    protocol = "org.apache.coyote.http11.Http11Protocol"
    port = "443"

    SSLEnabled = "true"
    scheme = "https"
    secure = "true"

    clientAuth = "false"

    sslProtocol = "TLS"
    sslEnabledProtocols = "TLSv1, TLSv1.1, TLSv1.2"
    ciphers = "... omitted for clarity; see below"

    keystoreFile = "${catalina.home}/conf/feistyduck.jks"
    keystorePass = "YOUR_PASSWORD"
    keyAlias = "server"

    sessionTimeout = "86400"
    sessionCacheSize = "10000"
/>
```

大部分参数的意义很直观，但请注意以下几点。

- 永远都不要修改SSLEnabled、scheme、secure和sslProtocol参数。
- 使用sslEnabledProtocols参数来控制协议选择（忽略sslProtocol参数，该参数供JSSE内部使用，无法提供有用的配置）。例子中没有启用SSLv2Hello和SSLv3，我认为是合理的，因为只有很古老的客户端才有需要，例如Windows XP上的IE6浏览器。
- 建议在Web服务配置中总是包含keystore设置，${catalina.home}可以用来避免使用绝对路径。
- keyAlias参数用于从密钥库中选择正确的证书链和密钥。
- 默认情况下，Tomcat不会限制TLS会话缓存数量，这可能会导致DoS攻击，最好是为TLS会话缓存设置一块固定大小的内存并进行相应配置。

配置示例中省略的部分是密码套件，建议使用以下默认配置：

```
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
```

① Tomcat and SSL Accelerators，http://blog.inuus.com/vox/2009/04/tomcat-and-ssl-accelerators.html（Paul Lindner的博客，2009年4月9日）。

```
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
```

以上选择是基于下面这些条件。

- 你使用的是Java 7，可以启用EC套件。
- 你没有使用DSA密钥（只支持1024位，服务器安全性较弱）。
- 你不希望使用不安全的DHE套件，因为它只支持768位DH参数。
- 同时加入ECDSA和RSA套件，无论用哪种密钥配置都能工作。

这套配置只支持前向保密以及强密码套件，在大部分现代浏览器和客户端上都可以正常工作，但某些古老的客户端可能会有问题。例如，Windows XP上的老IE版本访问会失败。

如果你确实需要为这些古老的客户端提供支持，当且仅当这种情况下，可以将下述套件添加到套件列表的最后：

```
TLS_RSA_WITH_AES_128_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA
TLS_ECDHE_RSA_WITH_RC4_128_SHA
SSL_RSA_WITH_RC4_128_SHA
```

前三个套件不在推荐列表中是因为它们不支持前向保密，后两个套件有同样的问题，并且还依赖不安全的RC4算法。记住，只有在仔细研究了RC4的弱点并有了足够理解后，你才可以判断是否使用RC4，强烈建议你从加密专家那里获取专业的意见。

> **注意**
> Java所支持密码套件的完整列表在SunJSSE Provider的文档中可以找到。[1]

1. 前向保密

以上推荐的套件配置并不能完全确保使用前向保密。主要有两个原因，都是由于Java 7和更早版本中JSSE的限制所导致的。

- JSSE不允许指定密码套件顺序，目前大部分客户端都会优先选择ECDHE套件（支持前向保密），但有些不是这样，其中一个例子是IE浏览器，直到最近为止，仍然会优先使用RSA套件而不是ECDHE套件。
- ECDHE是使用前向保密的首选，因为它的性能最好，然而不幸的是一些老客户端并不支持。必须启用DHE套件配置才能确保完全的前向保密。在JSSE中，DHE套件被限制只能使用不够安全的768位加密，因此在配置中并没有使用任何DHE套件，也就意味着这些老客户端将无法使用前向保密。

① JDK 8: The SunJSSE Provider，http://docs.oracle.com/javase/8/docs/technotes/guides/security/SunProviders.html#SunJSSEProvider（Oracle，检索于2014年7月17日）。

2. Java 8的配置

如果使用的是Java 8，那么你就自动得到了很多新功能。

- 更强的DH参数（1024位）默认被使用，但你应当通过配置jdk.tls.ephemeralDHKeySize系统属性将强度提高到2048位以加强安全性。如果你保留了默认配置，你的服务器可能很容易被国家级黑客破解。要了解有关DH安全状态的更多信息，请参阅6.5节。
- 可以配置JVM拒绝客户端重新协商。
- 使用默认套件配置的应用，可以自动获得并启用GCM密码套件。

除此之外的JSSE新特性，我们还需要多等一段时间直到Web服务器的升级启用，其中最重要的是以下两个特性。

- 遵循服务器密码套件优先。
- 支持虚拟安全主机。

以下是为Java 8应用推荐的密码套件配置：

```
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
TLS_DHE_RSA_WITH_AES_128_CBC_SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
```

推荐的列表更长了，不仅是因为加入了新的GCM套件，而且也加回了DHE套件，因为在Java 8中可以安全地使用DHE。

14.2.3 APR 和 OpenSSL 配置

要使用APR和OpenSSL组合来处理TLS，可以使用以下配置片段：

```
<Connector
    protocol = "org.apache.coyote.http11.Http11AprProtocol"
    port = "443"

    SSLEnabled = "true"
    scheme = "https"
    secure = "true"
```

```
    SSLVerifyClient = "none"

    SSLProtocol = "All"
    SSLCipherSuite = "……此处省略，描述见后"

    SSLHonorCipherOrder = "true"
    SSLCertificateFile = "${catalina.home}/conf/fd.crt"
    SSLCertificateKeyFile = "${catalina.home}/conf/fd.key"
    SSLCertificateChainFile = "${catalina.home}/conf/fd-intermediates.crt"
    SSLPassword = "KEY_PASSWORD"

    SSLDisableCompression = "true"
/>
```

与JSSE的配置相比，有些地方很类似而有些地方则存在差异。

- 协议选择有些问题，在我测试的版本（7.0.40）中，Tomcat无法感知TLS 1.1和TLS 1.2是否配置，也就是说SSLProtocol参数唯一实际可行的值就是All，SSL 3也就随之启用，而所有禁用SSL 3的相关努力也就失败了。Tomcat升级后，使用TLSv1+TLSv1.1+TLSv1.2的配置串可以解决问题。
- 与使用JSSE不同，没有控制SSL 2握手格式兼容性的办法，这个格式总是被支持。
- 可以通过SSLHonorCipherOrder参数来指定密码套件顺序。
- 没有密钥库的概念，密钥和证书以文件方式存储。
- 有一个参数可以禁用压缩，这个参数是必要的，因为OpenSSL是支持压缩的，这一点与JSSE不同（但是你应该永远禁用压缩，否则就可能会遭受CRIME攻击）。
- 没有控制TLS会话缓存的方法，这可能会让人感到不安。

对于推荐的密码套件配置，参考11.3.1节中的“推荐配置”部分；但是请注意，当前Tomcat Native还不支持ECDSA密钥。

全局OpenSSL配置

有些OpenSSL特性是全局配置的，需要通过AprLifecycleListener配置来控制，例如：

```
<Listener
    className = "org.apache.catalina.core.AprLifecycleListener"
    SSLEngine = "on"
    SSLRandomSeed = "builtin"
    FIPSMode = "off"
/>
```

在以下两种情况下你可能会需要对配置进行一些修改。

- 如果OpenSSL安装包支持多个引擎（例如硬件加速），可以在SSLEngine参数中指定所需的引擎名称。
- 如果OpenSSL安装包遵循FIPS，当需要启用FIPS模式时，请将FIPSMode参数设置为on。

第 15 章

配置Microsoft Windows和IIS

Microsoft是SSL/TLS和PKI生态系统的核心成员之一，其用户和操作系统（包括桌面系统和移动设备）无处不在，其服务器和云平台上运行着无数重要系统。大量网站都是在他们的开发环境下创建的。

Microsoft有着悠久的发展历史，其平台有着超长的生命周期，所以不难理解在SSL/TLS方面遇到的最大的问题是系统的复杂性以及缺乏良好的文档。复杂性来自于一个事实：Microsoft的软件代码库历史悠久，长期以来又不断加入各种新特性。文档则几乎没有，很难找到有用的信息。你常常需要在老旧的、目前已不准确的在线文档中查找。总而言之，Microsoft的密码库提供了对重要特性的良好支持，但也带着一些自己的特性。

15.1 Schannel

Microsoft安全通道（secure channel，通常简称为Schannel[①]）是一种加密组件，实现了一组用于安全通信的协议。Schannel是Windows平台上的官方SSL/TLS库，这意味着大多数的Windows程序都依赖它，特别是Microsoft自己开发的应用。

15.1.1 功能概述

Schannel一直以来都提供了对SSL和TLS协议功能的良好覆盖（参见表15-1）。Microsoft在2009年的Windows 7中第一个提供了对TLS 1.2的支持。相比较而言，OpenSSL在2012年才增加了对TLS 1.2的支持，而大多数桌面浏览器则是到2013年才开始支持。遗憾的是，虽然TLS 1.2功能已经实现，但默认设置却是禁用。更加讽刺的是，随后在2013年11月Microsoft发布IE11浏览器时又将TLS 1.2设为默认启用。

Microsoft的SSL/TLS实现上最大的问题是，Windows XP不支持虚拟安全托管；通过服务器名称指示（server name indication，SNI）扩展。我们无法指责Microsoft在2011年推出Windows XP时不支持SNI，因为SNI是2013年才制定的。但是，出于这样那样的原因，Microsoft在随后的3个Service

① Secure Channel，https://msdn.microsoft.com/en-us/library/windows/desktop/aa380123(v=vs.85).aspx（Microsoft Windows Dev Center，检索于2015年6月22日）。

Pack服务更新包中还是没有加入对SNI的支持，即使他们知道这个操作系统将使用很长时间。直到现在，Windows XP仍然拥有相当规模的用户数量，而缺乏SNI的支持使得部署大规模安全网站极为复杂和昂贵。好消息是，Windows XP SP3的支持在2014年4月已经终止，希望这些用户都能逐渐迁移到其他操作系统中。

> **注意**
> 本节描述了Schannel、Microsoft SSL/TLS库的功能。Schannel既继承了所依赖的底层库，又加入了自己的一些库，因为Windows采用了这种多层次的加密结构，有时候很难确定问题的来源是哪里。例如，虽然Windows 8的文档中宣称可以支持最大3072位的DSA密钥[①]，但Internet Explorer却不能与使用大于1024位密钥的服务器建立连接。这个限制就可能就存在于Schannel中。

表15-1　SChannel中SSL/TLS协议特性的发展

	Windows XP, Server 2003/ IIS 6	WindowsVista, Server2008/ IIS 7	Windows 7, Server2008 R2/ IIS 7.5	Windows 8, Server 2012 / IIS 8	Windows 8.1, Server2012 R2 / IIS 8.5
椭圆曲线加密	–	是	是	是	是
客户端SNI	–	是	是	是	是
服务器端SNI	–	–	–	是	是
TLS 1.0	可选	是	是	是	是
TLS 1.1、TLS 1.2[a]	–	–	是（IE11）[b]	是（IE11）[b]	是
AES套件	–[c]	是	是	是	是
AES GCM套件	–	–	有限支持[d]	有限支持[d]	有限支持[d]
大于1024位的DH参数	–	–	是（IE11）	是（IE11）	是[e]
会话票证	–	–	–	是（客户端）	是
安全重新协商	MS10-049	MS10-049	MS10-049	是	是
ALPN	–	–	–	–	是（客户端）
BEAST漏洞	MS12-006	MS12-006	MS12-006	是	是
OCSP stapling	–	–	是	是	是
默认客户端握手格式[f]	v2	v3	v3	v3	v3

a 此行展示的是IE浏览器默认设置，其他应用可能有不同的默认值，依赖于它们的具体SSL配置和使用的底层库。

b Windows 7加入了TLS 1.1和1.2的支持，但默认禁用，直到E11发布后启用。

c Windows Server 2003可以安装KB948963（2008年发布）以支持AES密码套件。

d Windows 10之前，已验证（authenticated，GCM）套件只支持ECDSA密钥交换，或者是DHE加RSA的组合。对于客户端通信来说，这意味着GCM套件在实践中很难被选用。

e 从Windows 8开始，DH参数可以支持最高4096位的安全性。

f 有两种客户单握手格式：老的用于SSL 2，新的由SSL 3引入。不是所有服务器都支持老格式，这意味着一些非常古老的客户端将无法访问这些服务器。

① BCryptGenerateKeyPair function，https://msdn.microsoft.com/en-us/library/windows/desktop/aa375451%28v=vs.85%29.aspx（Cryptography API: Next Generation documentation，检索于2014年2月4日）。

15.1.2 协议漏洞

由于拥有庞大的用户群（哪怕是很小的变化也可能影响极大的用户群，需要大量的测试），Microsoft在发现协议问题时的及时修复上有良好的记录。

- 不安全的重新协商
  与大部分其他厂商类似，Microsoft对不安全重新协商的早期解决方案也是直接禁用，对应的KB77377补丁在2010年2月9日发布[①]。随后在2010年8月10日对所有Windows平台发布了MS10-049[②]，完整地实现了安全重新协商（RFC5746）的功能。
- BEAST
  BEAST漏洞在2012年1月10日对所有Windows平台发布的MS12-006中被修复，使用1/*n*−1的数据包拆分方法来实现，这一方法也被TLS 1.0及更早版本的协议补丁所使用。
- CRIME
  Microsoft在自己的SSL/TLS协议栈中从未支持过TLS压缩，因此也就不会受到CRIME攻击的影响。
- Logjam
  在弱DH密钥交换的攻击响应上，Microsoft在2015年5月更新了客户端代码（例如IE浏览器），强制使用至少1024位的DH参数。[③]

15.1.3 互操作性问题

Schannel没有太多互操作性问题，需要注意的方面主要是弱和过时的加密算法。

- DSA
  Schannel不支持高于1024位的DSA密钥，也从未支持过。鉴于Microsoft用户群的广泛性，DSA名存实亡。DSA的密钥在加密强度上与RSA密钥大致相同，而1024位对于现有标准来讲强度太弱。在实践中这倒不是什么问题，因为在互联网上就几乎没有人在使用DSA密钥（也永远不会有）。
- 大于1024位的DH参数
  在版本11之前，IE浏览器不支持大于1024位的DH参数，但这个问题从未在实践中带来互操作性问题，因为IE浏览器并不支持DHE密钥交换和RSA身份验证的组合套件（互联网上主要使用的DH套件），2014年4月IE加入了DHE和RSA组合的套件支持，但只支持GCM变体，实际应用中几乎不会被协商到。
- 小于1024位的RSA密钥

① TLS/SSL漏洞可能允许欺骗，https://support.microsoft.com/zh-cn/kb/977377（Microsoft Security Advisory 977377，2010年2月9日）。

② SChannel中的漏洞可能允许远程代码执行，https://support.microsoft.com/zh-cn/kb/980436（Microsoft Security Bulletin MS10-049，2010年8月10日）。

③ SChannel中的漏洞可能允许信息泄露，https://support.microsoft.com/zh-cn/kb/3061518（KB 3061518，2015年5月12日）。

小于1024位的RSA证书和密钥的废弃最早是在2012年8月14日作为一个可选更新项发布，到2012年10月9日调整为强制更新[①]。这个更新对于公有和私有CA颁发的证书都是有效的。

- MD5

  2013年8月13日，Microsoft发布了KB2862973补丁[②]，宣布在Microsoft根证书计划范围内废除MD5签名，这一更新适用于Windows Vista、Server 2008以及其他更早的版本。Windows 8.1、RT 8.1和Server 2012 R2之后的平台，在操作系统发布时已经废除了MD5签名。

  由于这一更新只适用于根证书计划成员签发的证书，私有CA签发的MD5签名证书因此不受影响。如果需要完全废除MD5证书，可以手动安装KB2862966补丁[③]。

- RC4

  Microsoft是第一家废弃RC4的厂商，从Windows 8.1开始RC4加密算法就已经默认禁用了。2013年11月13日，Microsoft对Windows 8和之前的版本发布了KB2868725[④]，应用程序可以通过请求其他强加密算法来避免使用RC4，用户也可以通过修改注册表完全禁用RC4。

  IE11浏览器号称是第一个默认禁用RC4的浏览器[⑤]，虽然在Windows 8.1上这是真的，但在我的Windows 7桌面版（安装了2869725更新）上，RC4仍然可用。

  用户升级到IE11和Windows 8.1后将不再支持RC4，这有可能会导致互操作性问题。根据Microsoft的调查，2013年11月的采样中大约有3.9%的网站只支持RC4，SSL Pulse在2014年7月的调查数据是1.8%。在访问这些网站时，IE11在首次请求时会失败，之后它会自动降级重连，依次以TLS 1.0（仍然不支持RC4还是会失败）和SSL 3（这次有RC4了）建立连接。因此对于只支持RC4密码套件的网站，会发生以下两种情况：(1) 如果站点支持SSL 3，IE11最终会用这个协议版本成功连接，但在成功连接前会有一定延迟；(2) 如果网站不支持SSL 3，IE11就无法访问该网站。

  Microsoft不应该因为这个问题受到指责。作为一个拥有庞大用户群的厂商，第一个主动禁用一个常用密码套件，这是一个大胆的举动。从积极的一面来讲，对于只支持RC4的网站来讲，所受到的影响也会反过来促进网站运维来改进其自身的配置。

- SHA1

  2013年11月12日，Microsoft宣布计划于2016年底废弃SHA1签名证书[⑥]，同时，他们要求加

---

① 最小证书密钥长度更新，https://support.microsoft.com/zh-cn/kb/2661254（KB 2661254，2012年8月14日）。

② 该更新否决Microsoft根证书计划的MD5散列算法，https://technet.microsoft.com/library/security/2862973（KB 2862973，2013年8月13日）。

③ 改善Windows中弱证书加密算法管理的更新已发布，https://support.microsoft.com/zh-cn/kb/2862966（KB 2862966，2013年8月13日）。

④ 用于禁用RC4的更新，https://technet.microsoft.com/library/security/2868725（KB 2868725，2013年11月13日）。

⑤ IE11 Automatically Makes Over 40% of the Web More Secure While Making Sure Sites Continue to Work，http://blogs.msdn.com/b/ie/archive/2013/11/12/ie11-automatically-makes-over-40-of-the-web-more-secure-while-making-sure-sites-continue-to-work.aspx（IEBlog，2013年11月12日）。

⑥ SHA1 Deprecation Policy，http://blogs.technet.com/b/pki/archive/2013/11/12/sha1-deprecation-policy.aspx（Windows PKI blog，2013年11月12日）。

入Microsoft根证书计划的新成员必须提供4096位以上的SHA2签名的RSA密钥。在MD5问题上，Microsoft在终止日之后继续使用了MD5，从而遭遇到Flame恶意软件的攻击而广受批评。这次在SHA1问题上，Microsoft显然不希望重蹈覆辙。

除了以上列出的潜在问题，你需要考虑的Schannel相关最主要的互操作性问题，在于其仍然需要支持运行于古老的操作系统（例如SP3之前Windows XP）上的古老的客户端（例如IE6浏览器）。

15.2 Microsoft 根证书计划

Microsoft根证书计划（Microsoft Root Certificate Program）[①]维护着Windows操作系统中的可信证书集合。Windows Vista和更新版本的操作系统在交付时只部署了系统所必需的一小部分证书，其余的根证书会在第一次遇到时（例如，浏览网页时）安全地从Microsoft网站自动安装。基于这种实时更新机制，Microsoft的用户可以保证总是能持有最新的可信证书。

Windows XP不支持这种自动更新机制，需要安装系统更新来升级可信根证书，用户可以自己从Microsoft的更新列表中手动下载。[②]

15.2.1 管理系统可信证书库

如果运行的是新版的Windows系统，你基本不需要手动维护可信证书库，系统的自动更新机制会处理了所有的工作。可信证书列表每周自动更新一次，实际的根证书在使用时会自动按需下载，虚假证书和其他无效证书黑名单每天都会下载更新。[③]

> **注意**
>
> Windows实际运行着多个证书库。最主要的一个是计算机级别的，此外基于不同服务和不同用户账号都可以有独立的证书库。作为一个经验法则，最佳的工作方式是使用计算机证书证书库。

要查看和修改系统的可信证书库，可以使用Microsoft管理控制台（Microsoft management console，MMC），在15.5节中的“创建自定义IIS控制台”部分中会有详细说明。最核心的存库被称为可信根证书颁发机构（trusted root certification authority），它包含了Microsoft根证书计划中的所有根证书。默认情况下，该证书库中只包含了少量证书，但会根据使用情况自动更新，例如我的Windows桌面系统，在经过多年的使用后已经有了49个可信根证书。

① Introduction to The Microsoft Root Certificate Program，http://social.technet.microsoft.com/wiki/contents/articles/3281.introduction-to-the-microsoft-root-certificate-program.aspx（Microsoft TechNet Wiki，检索于2014年7月3日）。

② 配置可信根证书和不允许的证书，https://technet.microsoft.com/zh-cn/library/dn265983.aspx（Microsoft，检索于2014年7月2日）。

③ Announcing the automated updater of untrustworthy certificates and keys，http://blogs.technet.com/b/pki/archive/2012/06/12/announcing-the-automated-updater-of-untrustworthy-certificates-and-keys.aspx（Windows PKI Blog，2012年6月11日）。

如果你是一个Windows域管理员，那么可以使用组策略管理工具来合并整个域中的可信证书库。①

15.2.2　导入可信证书

添加一个新的可信CA很容易，在拿到正确的证书后，需要只是按证书导入向导操作。首先双击证书（扩展名应该是.cer），然后单击“导入证书”按钮即可完成导入。

> **警告**
> 添加一个新的可信CA前需要仔细衡量有无潜在的安全风险，一旦你信任了某个CA，也就代表你确信这家CA会正确地签发证书并且有良好的安全保障。请记住这一点，每一个CA都有能力签发世界上任意网站的证书②。

15.2.3　可信证书黑名单

由于Windows的自动更新机制，如果想撤销某个特定CA中的信任，简单地从可信根证书颁发机构（trusted root certification authority）证书库中删除其证书是不够的。如果这样做，系统在下次需要时会自动下载缺少的证书。

为了保证将证书永久加入黑名单，需要将它移动到不可信证书（untrusted certificate）证书库中。之后当你访问一个依赖该可疑根证书的网站时，IE浏览器（或其他依赖Windows可信证书库的程序）就会拒绝连接。

15.2.4　禁用根证书自动更新

如果你不喜欢根证书的自动更新机制，可以通过以下步骤来禁用。③

(1) 运行gpedit.msc打开本地组策略编辑器。(2) 在左窗格中，依次选择“计算机配置”→“管理模板”→“系统”→“Internet通信管理”→“Internet通信设置”。(3) 在右窗格中，找到并双击“关闭自动根证书更新”。(4) 要禁用自动更新，请将该设置更改为“已启用”。

请记住，从禁用自动更新的这一刻起，就必须手动地维护根证书。

15.3　配置

有趣的是对于一个图形界面为主的操作系统，Windows居然没有一个工具可以用来配置SSL/TLS协议、套件和加密算法。IIS自带了一个用于密钥和证书操作的简单用户界面，但是其他

---

① 管理可信根证书，https://technet.microsoft.com/en-us/library/cc754841.aspx（Windows Server 2012 documentation，检索于2014年7月3日）。

② 有一个被称为“名称约束”的特性，可以限制CA只能签发指定名称下的证书，但这个特性并未得到广泛使用。

③ Certificate Support and Resulting Internet Communication in Windows Vista，https://technet.microsoft.com/en-us/library/cc749331%28WS.10%29.aspx（Microsoft TechNet，检索于2014年7月3日）。

配置都需要直接修改注册表。

> **注意**
> 本节中的说明适用于操作系统以及使用系统库的程序，自带SSL/TLS和PKI库的程序不受影响，除非它们继承Schannel的配置。例如，Firefox浏览器使用了自己的库和根证书，Chrome浏览器虽然使用了自己的库但使用了系统的根证书。

15.3.1 Schannel 配置

Schannel配置可以用来调整协议和密码套件的使用。对于协议来说，客户端应用和服务器应用存在单独的控件；而对于其他一切，存在一组适用于所有应用类型的注册表项。

所有Schannel配置选项都嵌套在下面的根项之下。

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SecurityProviders\Schannel
```

1. 协议配置

协议是使用嵌套在`Protocols`子项下的大量注册表项进行配置的。每个协议版本都有一个自己的项，加上两个分别用于客户端和服务器应用的子项。从Windows Server 2008 R2和Windows 7开始，Windows支持从SSL 2.0到TLS 1.2之间的所有主要协议版本。以下是注册表的整体结构：[①]

```
Protocols\SSL 2.0
Protocols\SSL 2.0\Client
Protocols\SSL 2.0\Server
Protocols\SSL 3.0
Protocols\SSL 3.0\Client
Protocols\SSL 3.0\Server
Protocols\TLS 1.0
Protocols\TLS 1.0\Client
Protocols\TLS 1.0\Server
Protocols\TLS 1.1
Protocols\TLS 1.1\Client
Protocols\TLS 1.1\Server
Protocols\TLS 1.2
Protocols\TLS 1.2\Client
Protocols\TLS 1.2\Server
```

每个叶节点可以包含下面两个DWORD项之一或全部两个。

- ❑ `DisableByDefault`

  对于那些不明确指定协议版本并使用系统默认设置的应用，此项可以用来控制某个协议是否启用。如果项不存在或者值为`0`，则协议默认启用；如果值为`1`，则协议默认禁用。通常情况下，Windows会禁用SSL 2并启用所有其他协议。

- ❑ `Enabled`

  此项可以用来对所有应用强制禁用某个指定协议版本，即使应用明确选用了该协议也是

① 简便起见，列表中省略了部分子项：Multi-Protocol Unified Hello、PCT 1.0和DTLS 1.0，它们已经过时且极少使用。

如此。要强制禁用某个协议，请将Enabled项设置为0；如果这个项不存在或者值不为0（文档建议值是0xFFFFFFFF），则将启用协议。

对协议配置进行更改之后，需要重启所有活动程序才能使更改生效。

2. 密码套件算法选择

密码套件配置有两种配置方法。组成套件的加密算法可以单独进行配置。此外，如果某个特定算法被禁用，那么所有使用该算法的密码套件也将被禁用。这种机制确保了不会在任何地方使用弱算法，即使其他地方的配置建议使用弱算法也不会生效。

有以下子项可以设置，每种算法一个。①

```
Ciphers\AES 128
Ciphers\AES 256
Ciphers\DES 56
Ciphers\NULL
Ciphers\RC4 40/128
Ciphers\RC4 56/128
Ciphers\RC4 64/128
Ciphers\RC4 128/128
Ciphers\Triple DES 168
Hashes\MD5
Hashes\SHA
Hashes\SHA256
Hashes\SHA384
KeyExchangeAlgorithms\Diffie-Hellman
KeyExchangeAlgorithms\ECDH
KeyExchangeAlgorithms\PKCS
```

> **注意**
> PKCS密钥的配置仅适用于RSA密钥交换，RSA身份验证不受影响（例如，TLS_RSA_*套件会被禁用，但是TLS_ECDHE_RSA_*不会）。

要禁用某个算法，在对应的项下面创建一个名称为Enabled的DWORD项并设置为0，需要重新启用这个算法时，可以删除Enabled项或将值设为0xffffffff。有时配置更改会立刻生效，但为了确保新配置可靠加载，务必记得重新启动程序。

> **注意**
> 对散列的限制仅适用于密码套件，而不包含证书签名。如果要禁用MD5证书签名，需要按照本章后面的说明来操作。

15.3.2　密码套件配置

禁用单独的算法是很有用，但是大多数情况下，你真正想要的是明确指定启用的密码套件以及加载顺序。在Vista以及更新的系统上，Schannel允许这样的设置，并且这种调整会同时影响客

① 更老的Windows版本还支持RC2 40/128、RC2 56/128和RC2 128/128。

户端和服务器应用。

密码套件配置是唯一可以通过图形界面进行配置的Schannel设置。

(1) 首先，运行`gpedit.msc`启动本地组策略编辑器。[①]

(2) 在左窗格中，依次选择：“计算机配置”→“管理模板”→“网络”→“SSL配置设置”。

(3) 然后在右窗格中，双击“SSL密码套件顺序”并编辑修改。

> **警告**
> 通过策略编辑器修改密码套件配置时，请密切关注最终的套件字符串长度，编辑器只支持最大1023字节的数据并会在数据长度超出时自动截断。

在Microsoft的网站上可以找到Schannel支持的所有密码套件列表[②]，以下是我推荐的密码套件配置，着重考虑了安全性和性能：

```
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256_P256
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384_P384
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA_P256
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA_P256
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256_P256
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384_P384
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA_P256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA_P256
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256_P256
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
```

我作了以下假设。

- 只使用支持前向保密的套件。
- 同时支持RSA和ECDSA密钥，目前绝大多数网站使用的都是RSA密钥，所以ECDSA套件大部分情况下并不起作用，但好处是在未来需要迁移时不用再修改套件配置。
- 最后两个套件于2014年4月在Windows 8.1和Server 2012 R2中加入支持[③]，于2015年5月在Windows 7和Server 2008 R2中加入支持，实际环境中还不清楚是否会真正使用这两个套件，理论上讲支持这两个套件的客户端也支持更快的ECDHE套件，所以会优先选择前面的EC套件。
- 使用ECDHE、RSA和GCM组合的套件（例如`TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256_P256`）还不能使用，在本书写作期间（2015年7月），我们了解到Windows 10桌面系统已经支持这种组合，但Microsoft的服务器平台却不支持。当下一代Windows服务器平台发布

15

① 不是所有Windows操作系统都提供了这个工具，比如Windows 7专业版有，但是Windows 7家庭版没有。如果此工具不存在，可以直接修改注册表，本节后面部分有这方面的介绍。

② Cipher Suites in Schannel，https://msdn.microsoft.com/en-gb/library/windows/desktop/aa374757%28v=vs.85%29.aspx（Microsoft，检索于2014年7月17日）。

③ KB 2929781: Update adds new TLS cipher suites and changes cipher suite priorities in Windows 8.1 and Windows Server 2012 R2，https://support.microsoft.com/en-us/kb/2929781（Microsoft，2014年4月8日）。

时，或者会重新引入这种组合套件，你应该将它们放在其他RSA套件之前，即对RSA算法优先使用这种组合。

上述配置只选用了支持前向保密和强加密的套件。大部分现代浏览器和其他客户端都能成功建立连接，但一些老客户端可能会有问题。例如，运行于Windows XP上的老版本IE浏览器就会连接失败。

如果你确实需要为这些老客户端提供支持（并且只有在这种情况下），可以将以下套件添加到密码套件列表的尾部：

```
TLS_RSA_WITH_AES_128_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA
TLS_RSA_WITH_RC4_128_SHA
```

前三个套件不在推荐列表中是因为它们不支持前向保密，最后的套件有同样的问题，并且还依赖不安全的RC4算法。记住，只有在仔细研究了RC4的弱点并充分理解后，你才可以判断是否使用RC4，强烈建议你从加密专家那里获取专业的意见。

> **注意**
> 如果你仔细查看套件的名称，就会发现Microsoft使用了扩展密码套件名称的语法，通过正式名称（比如TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256）和一个P256或P384的后缀，这两个后缀指定了用于ECDHE密钥交换的椭圆曲线算法，分别代表NIST的secp256r1和secp384r1曲线模型。尽管两种后缀实际使用的底层套件名称相同，这种命名方法可以让你自主选择优先使用哪种曲线模型。

如果你想通过直接操作注册表来配置密码套件，对应的注册表项是：[①]

```
HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Cryptography\Configuration\SSL\00010002
```

如果该项为空，可以创建一个新项，即类型为MULTI_SZ的Functions（字符串列表）。将值设置为需要默认启用的密码套件列表并按优先顺序排序。直接使用注册表编辑器进行此修改很容易，如果想要从命令行或者注册表文件导入，请将所有套件放在一行内并用逗号隔开，注意不要插入任何空格。注册表修改完成后，需要重新启动系统才能生效。

15.3.3　密钥和签名限制

不久前，Microsoft提供了在证书链验证时对弱加密算法限制的功能。这一特性在Windows 8.1和Windows Server 2012 R2之后已默认提供，更早的Windows系统可以通过安装KB2862966补丁[②]

---

① 有多个项可以影响密码套件的配置，有些文章推荐使用HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Cryptography\Configuration\Local\SSL\00010002，但此项优先级较低，在有其他配置时可能失效。此外，也应当尽量避免直接操作注册表。

② Windows弱证书加密算法管理改进的更新，https://support.microsoft.com/en-us/kb/2862966（KB 2862966，2013年8月13日）。

获取。

Microsoft的策略框架具有良好的扩展性并且支持很多有用的功能。

- ❑ 禁用弱加密算法
- ❑ 对于密钥算法，强制最小密钥长度
- ❑ 根据证书类型执行不同的策略（例如，对于服务器身份验证和代码签名，可以使用不同的策略）
- ❑ 指定策略应用于所有证书或者只针对公共CA
- ❑ 指定策略应用于某个特定日期之后签发的证书（例如，兼容老证书，但不允许新证书使用弱算法）
- ❑ 记录策略违规行为
- ❑ 记录违规行为，但除此之外并不强制执行策略
- ❑ 创建每证书例外

推荐你先只启用策略的日志功能，观察违反策略的场景，避免因为预想与实际不一致而导致潜在的损失。通过不断调整和观察实际情况，最后可以相对安全地启用策略的强制实施。一旦一个策略在某个机器上测试通过，就可以通过组策略对象将它推送到整个网络。

在本书写作时，已经可以限制DSA、ECDSA和RSA密钥中MD5和SHA1签名的使用，这可以通过修改以下注册表项下的内容来实现：

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\OID\EncodingType 0\ ↩
CertDllCreateCertificateChainEngine\Config
```

因为策略可以很复杂，Microsoft通过特殊的命名方法来表示存储在注册表中的规范，每个项名称必须采用以下格式：

```
Weak<CryptoAlg><ConfigType><ValueType>
```

要构建一个项名称，请用表15-2中对应选项的值替换掉名称模板中的选项。

表15-2　注册表项名称模板选项

选　项	值	描　述
CryptoAlg	Md5 Sha1 Dsa Ecdsa Rsa	指定策略所应用到的算法的名称
ConfigType	ThridParty	应用于Microsoft根证书计划的公共CA
	All	应用于所有根证书（公共和私有CA），因为ThirdParty是All的子集，所以也适用以下条件 ❑ 在All上设置的大部分标志也会在ThirdParty上设置，日志记录标志不会受到影响 ❑ 证书签发时间以最早的（AfterTime）为准 ❑ 密钥长度以最大的（MinBitLength）为准

（续）

选　　项	值	描　　述
ValueType	Flags	用于选择哪些证书类型受限制以及如何受限制的标志列表，详见后面的说明（REG_DWORD）
	MinBitLength	指定密钥算法的最小公钥长度（REG_DWORD）
	AfterTime	策略仅对指定日期后签发的证书有效，时间戳证书链除外（REG_BINARY类型的一个8字节FILETIME）
	Sha256Allow	明确允许的弱证书列表，以十六进制的SHA256指纹表示（REG_SZ或REG_MULTI_SZ）

项标志有双重含义，首先它们被用来指定规则是否启用和生效，参见表15-3。

表15-3　用于规则启用和生效的标志

标　　志	描　　述
CERT_CHAIN_ENABLE_WEAK_SETTINGS_FLAG（0x80000000）	此标志用于指定策略是否启用，如果未启用，则同一个CryptoAlg和ConfigType组合的其他设置都无效
CERT_CHAIN_ENABLE_WEAK_LOGGING_FLAG（0x00000004）	此标志用于指定策略日志是否启用，即为策略违规生成日志
CERT_CHAIN_ENABLE_ONLY_WEAK_LOGGING_FLAG（0x00000008）	此标志用于指定策略是否仅启用日志功能，即对策略违规只做日志记录，不实际处理。这个设置在实际启用策略前的测试中非常有用

其次，其余标志用于指定规则应用于哪些证书类型，详见表15-4。

表15-4　用于指定规则对应证书类型的标志

标　　志	描　　述
CERT_CHAIN_DISABLE_ALL_EKU_WEAK_FLAG（0x00010000）	策略应用于所有证书类型
CERT_CHAIN_DISABLE_SERVER_AUTH_WEAK_FLAG（0x00100000）	策略应用于服务器身份验证证书
CERT_CHAIN_DISABLE_CODE_SIGNING_WEAK_FLAG（0x00400000）	策略应用于代码签名证书
CERT_CHAIN_DISABLE_MOTW_CODE_SIGNING_WEAK_FLAG（0x00800000）	策略应用于从网站下载的代码签名证书
CERT_CHAIN_DISABLE_TIMESTAMP_WEAK_FLAG（0x04000000）	策略应用于时间戳证书
CERT_CHAIN_DISABLE_MOTW_TIMESTAMP_WEAK_FLAG（0x08000000）	策略应用于从网站下载的时间戳证书

> **注意**
> 要指定一个弱签名算法策略，请在对应的注册表项中启用CERT_CHAIN_ENABLE_WEAK_SETTINGS_FLAG（例如，对于MD5是WeakMd5AllFlags）。要指定一个弱密钥算法策略，启用相应的标志并配置最小密钥长度（例如，如果需要屏蔽所有小于1024位的RSA密钥，请将WeakRsaAllMinBitLength设置为1024）。

1. 使用CertUtil来管理加密策略

直接操作注册表有时候会非常棘手，这是必然的，因为策略实在是相当复杂。另一种维护策略的方法是使用CertUtil工具，它能够展示、创建、修改、删除策略的注册表项，也可以单独修改其中的标志、时间和字符串列表。

```
$ CertUtil -setreg -?
用法：
  CertUtil [选项] -setreg [{ca|restore|policy|exit|template|enroll|chain|PolicyServers}\↩
[ProgId\]]RegistryValueName值
  设置注册表值
    ca -- 使用CA的注册表项
    restore -- 使用CA的还原注册表项
    policy -- 使用策略模块的注册表项
    exit -- 使用第一个退出模块的注册表项
    template -- 使用模板注册表项（对用户模板使用-user）
    enroll -- 使用注册注册表项（对用户上下文使用-user）
    chain -- 使用链配置注册表项
    PolicyServers -- 使用策略服务器注册表项
    ProgId -- 使用策略或退出模块的ProgId（注册表子项名称）

    RegistryValueName -- 注册表值名称（使用"Name*"来作为匹配值的前缀）
    Value -- 新的数字、字符串或日期注册表值或文件名。
      如果一个数值以"+"或"-"开始，在新的值中指定的位将在
      现有的注册表值中被设置或清除

      如果字符串值以"+"或"-"开始，并且现有值为
      一个REG_MULTI_SZ值，此字符串将添加到
      现有的注册表值中，或从中删除
      要强制创建一个REG_MULTI_SZ值，请在字符串值的结尾添加
      一个"\n"

      如果值以"@"开始，则值的其余部分为
      文件的名称，该文件包含
      代表一个二进制值的十六进制文本
      如果它未引用一个有效文件，则会将其作为
      [Date][+|-][dd:hh]进行分析，可选日期加上或减去可选
      天数和小时数
      如果同时指定两者，则使用加号（+）或减号（-）分隔符
      将"now+dd:hh"用于相对于当前时间的日期

  使用"chain\ChainCacheResyncFiletime @now"有效地刷新缓存的CRL

选项：
  -f                -- 强制覆盖
  -user             -- 使用HKEY_CURRENT_USER项或证书库
  -GroupPolicy      -- 使用组策略证书库
  -gmt              -- 将时间显示为GMT
  -seconds          -- 用秒和毫秒显示时间
  -v                -- 详细操作
  -privatekey       -- 显示密码和私钥数据
  -config Machine\CAName    -- CA和计算机名称字符串

CertUtil -?              -- 显示谓词列表（命名列表）
CertUtil -setreg -?      -- 显示"setreg"谓词的帮助文本
CertUtil -v -?           -- 显示所有谓词的所有帮助文本
```

> **警告**
> 如果你使用CertUtil工具修改注册表项，对于加密策略的修改会立刻生效。老生常谈，建议在修改前备份好注册表。

2. 记录弱证书链

可以记录弱证书链用于日后分析。要启用这个特性，请使用所需的证书库位置配置WeakSignatureLogDir项：

```
$ CertUtil -setreg chain\WeakSignatureLogDir C:\Log\WeakCertificateChains
```

此外，需要确保策略中的CERT_CHAIN_ENABLE_WEAK_LOGGING_FLAG标志有效，或者在希望只记录而不实际执行策略时，设置CERT_CHAIN_ENABLE_ONLY_WEAK_LOGGING_FLAG标志。

3. 完整的策略示例

为了详细说明，我会展示一个简单的策略。该策略会使用日志记录对包含以下内容的任何证书强制实施限制。

- MD5签名
- 小于1024位的RSA密钥
- 小于1024位的DSA密钥
- 小于160位的ECDSA密钥

初始策略将假定只进行日志记录而不强制实施：

```
$ CertUtil -setreg chain\WeakSignatureLogDir C:\Log\WeakCertificateChains
$ CertUtil -setreg chain\WeakMd5AllFlags 0x80010008
$ CertUtil -setreg chain\WeakRsaAllFlags 0x80010008
$ CertUtil -setreg chain\WeakRsaAllMinBitLength 1024
$ CertUtil -setreg chain\WeakDsaAllFlags 0x80010008
$ CertUtil -setreg chain\WeakDsaAllMinBitLength 1024
$ CertUtil -setreg chain\WeakEcdsaAllFlags 0x80010008
$ CertUtil -setreg chain\WeakEcdsaAllMinBitLength 160
```

0x80010008这个值代表了以下三个标志的组合：

```
CERT_CHAIN_ENABLE_WEAK_SETTINGS_FLAG (0x80000000)
CERT_CHAIN_DISABLE_ALL_EKU_WEAK_FLAG (0x000010000)
CERT_CHAIN_ENABLE_ONLY_WEAK_LOGGING_FLAG (0x000000008)
```

对应的注册表文件是：

```
Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\OID\EncodingType 0\ ↩
CertDllCreateCertificateChainEngine\Config]
"WeakSignatureLogDir"="C:\\Log\\WeakCertificateChains"
"WeakMd5AllFlags"=dword:80010008
"WeakRsaAllFlags"=dword:80010008
"WeakRsaAllMinBitLength"=dword:00000400
"WeakDsaAllFlags"=dword:80010008
"WeakDsaAllMinBitLength"=dword:00000400
"WeakEcdsaAllFlags"=dword:80010008
"WeakEcdsaAllMinBitLength"=dword:000000a0
```

当希望将策略从纯日志模式调整为正式启用时，可以将0x8001008更改为0x80010004（将CERT_CHAIN_ENABLE_ONLY_WEAK_LOGGING_FLAG替换为CERT_CHAIN_ENABLE_WEAK_LOGGING_FLAG），重新加载配置文件。或者可以通过以下命令单独调整：

```
$ CertUtil -setreq chain\WeakMd5Flags -0x00000008
$ CertUtil -setreq chain\WeakMd5Flags +0x00000004
```

15.3.4 重新协商配置

关于重新协商，Windows系统有两三个可以自定义配置的地方，其中最重要的是对安全重新协商的支持。你应该在所有服务器和个人工作站上都启用这一特性，Windows 8之前的版本需要安装MS10-049补丁。

然而，仅仅在服务器上启用安全重新协商并不能从根本上完全解决问题，出于兼容性的考虑，大部分服务器的配置仍然支持不安全的客户端重新协商请求，在MS10-049补丁中这被称为兼容重新协商（compatible renegotiation）。在这种模式下，当客户端或者服务器发起重新协商请求时，即使无法达成安全重新协商，Schannel也不会直接终止连接。

如果你不需要服务器的重新协商特性，解决问题就很容易了。在正式支持安全重新协商之前，Microsoft发布过一个过渡补丁KB977377，允许用户直接禁用重新协商。如果你在服务器禁用了重新协商，客户端是否支持安全重新协商也就无所谓了。要禁用重新协商，请将以下项设置为一个非0的值。

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\SecurityProviders\SCHANNEL\DisableRenegoOnServer
```

> **注意**
> 早期的IIS版本支持客户端发起的重新协商，但从IIS6及之后的版本就不允许了。严格来讲，如果你的服务器从不发起重新协商（如果你没有使用客户端证书验证），重新协商的安全漏洞就不会被真正利用。尽管如此，我仍然建议另外再明确禁用一下重新协商功能，因为其他程序仍然有可能受到漏洞影响。例如，Microsoft的Forefront threat management gateway（TMG）就允许客户端发起的重新协商。

从另一方面讲，如果你需要启用服务器发起的重新协商，那么唯一可选择的是切换到严格重新协商（strict renegotiation）。在这种模式下，你的服务器将只接受客户端发起的安全重新协商请求，在安全性上是有保障的，但是有可能导致未修复漏洞的浏览器请求被拒绝。

要启用严格重新协商，需将以下项设置为0。

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\SecurityProviders\SCHANNEL\ ↩
AllowInsecureRenego Clients
```

在我的测试中，这个配置更改会立即生效，无需重新启动应用程序。

最后一个需要你作出判断的是，对于你自己的客户端（例如浏览器），是否允许它连接到那些不支持安全重新协商的服务器。默认情况下是允许的，但是这会有风险，因为服务器有可能已被入侵。如果要确保安全，在客户端中启用强制安全重新协商，代价与服务器类似，你将无法连

接到相当一部分网站上。根据SSL Pulse在2014年7月的报告，大约有11.6%的服务器还不支持安全重新协商。

如果你决定将客户端切换到严格重新协商模式，请将以下项设置为0。

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\SecurityProviders\SCHANNEL\ ↩
AllowInsecureRenego Servers
```

> **注意**
> KB 977377的过渡方案也适用于客户端的重新协商禁用，但在安全性上这并无改进，不安全的重新协商漏洞利用的是欺骗服务器接受错误的重新协商请求，不影响客户端。

15.3.5　配置会话缓存

SSL和TLS允许使用会话缓存来避免每次连接时重复执行最消耗资源的加解密操作，对此Schannel会在服务器内存中维护一份会话信息缓存。注意在不同平台上有不同的默认设置，因此你最好在所有服务器上都做好明确的配置。[①]

所有会话缓存的参数都位于Schannel的主注册表项下：

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SecurityProviders\Schannel
```

- 需要配置服务器会话缓存的保留时间，请将ServerCacheTime设置为你需要的时间，单位是毫秒。
- 一般情况下无需为客户端应用设置会话缓存时间，如果确有需要，请设置Client-CacheTime，单位也是毫秒。
- 需要配置可保存的会话数量上限时，创建或修改MaximumCacheSize参数的值。如设置为0则表示禁用会话缓存。

按一般规律来讲，你应该为会话缓存分配尽量多的内存，理想情况下，所有会话都可以缓存在内存中直到过期（而不会因为内存不足被清理）。每个会话需要占用2~4 KB的内存，因此想要得到最大会话缓存数量，可以用你规划的内存空间除以4 KB来计算。

然而这种方法有一个问题，Schannel的会话缓存实现方式会导致内存使用超出预计的上限。其原因在于新的会话缓存是实时生成的，而过期的会话则是定期清理的（以ServerCacheTime设置的时间为间隔），无论内存是否已达到使用上限。在正常的流量下，即使有高峰期，这种工作方式一般也没问题，但是这确实带来了一个新的DoS攻击点。例如，攻击者可以每秒不断地创建大量SSL会话，这些会话在一段时间内全部都会保存在内存中（每个消耗4 KB），直到缓存被修剪。

通常我会建议将会话缓存保留时间设为24小时，但是考虑到Schannel的上述缓存机制，有理由使用一个短得多的时间，比如1小时，并考虑为缓存分配更多的内存作为缓冲。

---

① 如何配置安全套接字层服务器和客户端缓存元素，https://technet.microsoft.com/library/security/2868725（Microsoft技术支持网站，2008年7月7日）。

> **注意**
>
> 从Windows 8.1开始，Schannel支持服务器的**会话票证**（session ticket），这是一种无状态的会话恢复机制。然而，直到本书写作期间，这一特性还没有正式公开，PowerShell的文档中有一些相关的信息。[1]

15.3.6 监控会话缓存

Schannel中开放了数个性能计数器用于监控会话缓存情况和会话恢复成功率。在那些老的平台上，恢复率只受服务器会话缓存影响。此外会话票证对提高恢复率应该有良好的效果，假如平台支持这一特性的话。

相关的性能计数器（详见表15-5）都位于Security System-Wide Statistics类别下，你可以用性能监控工具来查看（在命令行上运行`perfmon`）。

表15-5 SChannel性能计数器

性能计数器	描 述
Schannel会话缓存活跃数量（Active Schannel Session Cache Entries）	此计数器用于记录Schannel会话缓存中存储的所有有效SSL会话数量。Schannel会话缓存中记录了已成功建立连接的会话信息，例如SSL会话ID。客户端可以用这些信息重新链接到服务器而无需重复执行完整的SSL握手
Schannel会话缓存总数量（Schannel Session Cache Entries）	此计数器用于记录Schannel会话缓存中存储的所有SSL会话数量。Schannel会话缓存中记录了已成功建立连接的会话信息，例如SSL会话ID。客户端可以用这些信息重新链接到服务器而无需重复执行完整的SSL握手
SSL客户端完整握手（SSL Client-Side Full Handshakes）	此计数器用于记录客户端每秒中执行的SSL完整握手数量。握手过程指的是计算机或其他设备之间的通过数据交换来确认通信是否可以建立
SSL客户端重连握手（SSL Client-Side Reconnect Handshakes）	此计数器用于记录客户端每秒中执行的SSL重连握手数量。重连握手允许恢复已建立的SSL会话，与完整握手相比，资源消耗很少
SSL服务器完整握手（SSL Server-Side Full Handshakes）	此计数器用于记录服务器每秒中执行的SSL完整握手数量。握手过程指的是计算机或其他设备之间的通过数据交换来确认通信是否可以建立
SSL服务器重连握手（SSL Server-Side Reconnect Handshakes）	此计数器用于记录服务器每秒中执行的SSL重连握手数量。重连握手允许恢复已建立的SSL会话，和完整握手相比资源消耗很少

15.3.7 FIPS 140-2

联邦信息处理标准（federal information processing standard，FIPS）是由美国国家技术研究院（National Institute of Standards and Technology，NIST）开发的一组标准，用于非军事用途的政府系统。其中包括了各种各样的标准，并非全部都与安全性相关，而与安全性有关的标准中，FIPS 140-2是我们最感兴趣的，因为它确定了加密技术的使用指南。为简单起见，后文中我会把FIPS 140-2简称为FIPS。

为美国政府设计的系统都需要遵循FIPS标准。通常来讲，要确保符合标准相当复杂：首先，

[1] Transport Layer Security Cmdlets in Windows PowerShell，https://technet.microsoft.com/library/security/2868725（Microsoft TechNet，2013年10月17日）。

你需要确保系统运行的全部是合法的加密组件；其次，对于所有部署的应用都需要确保符合加密标准。

Microsoft使这个过程变得相对简单，因为它维护了核心库和组件的标准兼容性，因此最难的就变成了要确保第三方程序以及自己开发的程序都符合加密标准。

在所有Windows平台上，FIPS的有效实施可以分为以下五个层面。

- 底层库

  Microsoft一直积极地为两个核心加密库维护着FIPS 140认证：加密API（cryptographic API，CAPI）以及下一代加密API（cryptographic API: next generation，CNG）。这两个底层库本身并不关心FIPS，它们同时提供已审批的和未审批的加密算法，保障系统遵循标准是上层系统的职责。

- FIPS注册表项

  有一个注册表项用来标示当前系统是否需要符合FIPS标准，所有在系统上部署的应用都需要根据这个参数来决定自身的行为是否需要符合标准。

- 上层库

  关心FIPS的上层加密库，需要根据FIPS注册表项的参数来调整自己的行为。这里特别强调一下，Schannel和Microsoft的.NET框架完全遵循这一FIPS参数。

- 操作系统组件

  核心操作系统组件都宣称依赖和支持FIPS，这使得FIPS部署简单了许多。例如，远程桌面协议（remote desktop protocol，RDP）、文件加密系统（EFS，BitLocker）以及IPSec都是FIPS兼容的。

- 应用

  应用是加密算法的实际消费者，因此在FIPS兼容性上具有最终的责任。基于底层库的应用（CAPI和CNG），需要做一些额外的工作来确保底层库的使用符合标准。而另一方面，只依赖上层库的应用默认就是标准兼容的。

配置FIPS

启用FIPS最简单的方法是使用本地安全策略管理控制台。

(1) 从命令提示符或者“运行”菜单中，调用`secpol.msc`。

(2) 在左窗格中，依次选择：“本地策略”→“安全选项”。

(3) 在右窗格中，找到并双击“系统加密：将FIPS兼容算法用于加密、散列和签名”项。

(4) 在弹出的属性窗口中，选择“已启用”或“已禁用”，然后按“应用”（如图15-1所示）。

> **注意**
>
> 更改FIPS相关设置后需要重启系统。



图15-1 使用本地安全策略管理控制台配置FIPS

如果你喜欢直接操作注册表，你需要将FIPS注册表项的值设置为1（启用）或者0（禁用）。FIPS注册表项的位置与操作系统有关。在Windows Vista和之后的版本中，该注册表项位于：

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa\FIPSAlgorithmPolicy\Enabled
```

在Windows XP和Windows Server 2003上，该注册表项位于：

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa\FIPSAlgorithmPolicy
```

15.3.8 第三方工具

也许你了解所有Schannel的注册表项，但这并不代表你每次都想直接修改注册表。Nartac软件公司的IIS Crypto（详见图15-2）是一个IIS配置工具，允许你配置密码套件的启用和顺序，它配备了预定义的模板，以及一个到SSL Labs网站的快捷链接，你可以用它来测试验证自己的新配置。

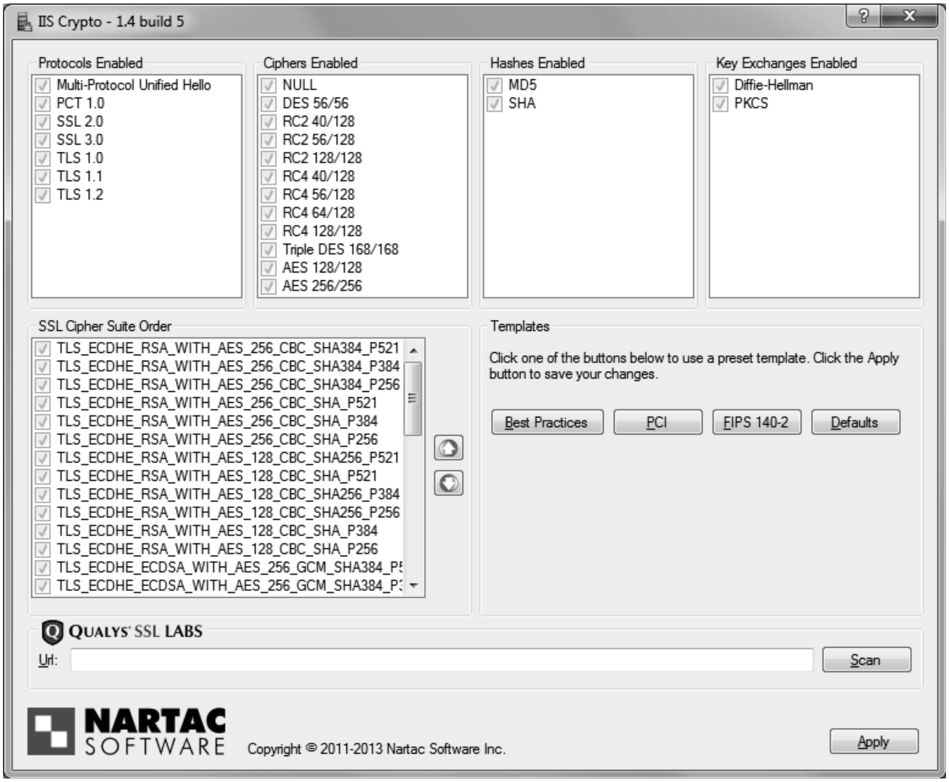


图15-2　Nartac软件公司的IIS Crypto配置工具

15.4　保护 ASP.NET 网站应用的安全

这一节将讨论如何安全部署ASP.NET应用的主题，包括应用有哪些不安全的实现方式，例如允许明文访问或者使用不安全的Cookie。

15.4.1　强制使用 SSL

为了防止误配置带来问题，需要运行于TLS之下的应用，应该对每个请求加上主动的TLS检验，代码如下：

```
if (Request.Url.Scheme.Equals("https") == false) {
    // 错误，请求未使用SSL
}
```

然而，让每个执行单元（脚本）都单独加上SSL检查通常不太可行。一个更好的方法是只写一段代码然后在需要的地方直接调用。ASP.NET支持授权过滤器，用于在每次请求时执行特定的代码。这种过滤器是TLS检查代码实现的理想方式。

15.4.2　Cookie 的保护

应用程序使用的每一个Cookie都应该单独进行保护，你需要做的就是将Secure属性设置为true，如果不打算从JavaScript访问Cookie，那么将HttpOnly属性也设置为true。

```
// 新建一个Cookie并初始化
HttpCookie cookie = new HttpCookie();
cookie.Name = "CookieName";
cookie.Value = "CookieValue";
cookie.Expires = DateTime.Now.AddMinutes(10d);

// 设置Cookie的安全属性
cookie.HttpOnly = true;
cookie.Secure = true;

// 将Cookie加入响应头
Response.Cookies.Add(cookie);
```

15.4.3　保护会话 Cookie 和 Forms 身份验证的安全

ASP.NET配置文件中的<httpCookies>元素[①]用于控制会话Cookie的安全性，例如，要设置会话Cookie的httpOnly标志（禁止JavaScript访问会话Cookie）和Secure标志（确保Cookie只在SSL时传送），配置如下所示。

```
<configuration>
    <!-- 其他配置选项 -->

    <system.web>
        <httpCookies
            domain = "www.example.com"
            httpOnlyCookies = "true"
            requireSSL = "true"
            lockItem = "true"
        />
    </system.web>
</configuration>
```

lockItem属性的作用是避免其他地方的配置覆盖了这里配置的值。尽管如此，仍然有一个问题：如果你的配置还包括了<form>元素（换句话说，就是使用了Forms身份验证），需要确保<forms>的requreSSL属性也设置为true。

```
<forms
    requireSSL = "true"
    cookieless = "UseCookies"
    <!—其他属性 -->
/>
```

15

① httpCookies元素，https://msdn.microsoft.com/zh-cn/library/ms228262(v=vs.100).aspx（.NET Framework 4 documentation，检索于2014年7月3日）。

你会发现我还将cookieless属性设置成了UseCookies，Forms身份验证支持两种方式来传输会话令牌：最常用的是使用Cookie，但也可以通过URI参数的方式将会话令牌添加到页面链接中。基于URI的方法很有意思，即使应用程序不支持Cookie也能工作，然而它也会带来明显的安全问题：浏览器在通过链接跳转时会将当前URI添加到Refer请求头中，因此会将会话令牌暴露给其他网站；如果攻击者成功引导用户进入一个攻击者控制的网站，他就可以劫持用户的会话。

15.4.4 部署 HTTP 严格传输安全

HTTP严格传输安全（HTTP strict transport security，HSTS）是一个较新的标准，允许Web服务程序要求浏览器只使用安全的访问方式。这一事实也使HSTS成为了一个底层防御机制，即使应用程序的设计有误（例如使用了不安全的会话Cookie），也能起到保护作用。另外，无效证书的处理得到了加强，用户再也无法跳过浏览器的安全警告而继续访问。部署HSTS很简单，但在此之前你需要充分了解它的优缺点。

以下代码样例启用了HSTS并将有效期设置为大约一年（以秒为单位），对于主域名和所有子域名同时生效：

```
Response.AppendHeader(
    "Strict-Transport-Security",
    "max-age=31536000; includeSubDomains; preload"
);
```

你也可以直接在配置文件中添加HTTP响应头，配置如下：

```
<configuration>
    <!—其他配置选项 -->

    <system.webServer>
        <httpProtocol>
            <customHeaders>
                <add name="Strict-Transport-Security"
                     value="max-age=31536000; includeSubDomains; preload" />
            </customHeaders>
        </httpProtocol>
    </system.webServer>
</configuration>
```

IIS的管理界面上也支持自定义响应头的设置。然而使用以上任意方法时都有一个问题，即HSTS规范不允许在明文响应中返回Strict-Transport-Security响应头。因此最简单和最干净的方案是使用第三方模块，将具体细节全部交由模块处理。[1]

15.5 Internet 信息服务

Internet信息服务（internet information services，IIS）是Windows操作系统上最主要的Web服

① HTTP Strict Transport Security IIS Module，http://hstsiis.codeplex.com/（CodePlex，检索于2014年7月2日）。

务器，它有多种发行版本（例如桌面版和服务器版），但底层代码基本一致，并且所有版本在SSL/TLS特性上最终都依赖Schannel。

因为Schannel是一个相当成熟的TLS库，IIS也自然对TLS有了良好的支持。实践应用中最大的问题在于IIS没有用于TLS配置的用户界面，而完全依赖Schannel的配置，Schannel反过来又只能通过直接操作注册表来配置，这就带来了不少困难。

在这一节的剩余部分，我将重点讨论在IIS上运行安全网站的一些问题。

- 前向保密

  IIS支持使用ECDHE密钥交换的前向保密，适用于大多数客户端。由于Schannel不支持DHE和RSA组合的密码套件，就无法给哪些不支持椭圆曲线算法的老客户端提供DHE密钥交换方式，而这对你的系统的影响程度，完全取决于你的用户群。例如Twitter曾经报道说大约有25%的用户不支持ECDHE。[①]

  2014年4月，Microsoft在Windows 8.1和Server 2012 R2上发布了一个更新包，增加了2个新的`DHE_RSA`套件（使用1024位DH参数）支持，然而这两个套件在改善老客户端前向保密的支持上于事无补，因为它们只支持GMC验证加密，而这个老客户端通常也不支持。2015年5月，Windows 7和Windows Server 2008 R2的系列版本中也加入了对这两个套件的支持。[②]

  此外，新增加的套件仍然在使用1024位DH密钥交换参数，现在的安全性已经较弱，也没有其他办法来配置IIS以支持更强的DH参数。要了解有关DH安全状态的更多信息，参阅6.5节。

- GCM套件

  在本书写作时，GCM套件被认为是唯一完全安全的套件。即使其他套件的问题基本上都得到了解决，假如你特别热衷于实现最佳的安全性，GCM套件就是你的最优选择。Schannel确实支持GCM套件，但通常都是与ECDSA密钥绑定。ECDSA密钥的问题在于不是所有客户端都能支持。例如，直到2015年7月，大约有6%的维基百科用户还不支持ECDSA密钥。[③]

- OCSP stapling

  从Windows 2008开始，IIS已经默认启用OCSP stapling。由于大部分其他Web服务器在这方面都需要手动配置，导致互联网上绝大多数的stapling响应都来自IIS服务[④]。唯一的缺点是IIS服务器必须能与签发证书的CA通信来获取OCSP响应和本地缓存。如果系统环境在出口流量上有严格的控制（防火墙），这种请求就有可能被阻止。要解决这个问题，你可以放宽防火墙的策略或者对OCSP请求配置一个转发代理。[⑤]

---

① Forward Secrecy at Twitter，https://blog.twitter.com/2013/forward-secrecy-at-twitter（Twitter’s Engineering Blog，2013年11月22日）。

② 默认密码套件优先级顺序更新，https://technet.microsoft.com/zh-cn/library/security/3042058.aspx（Microsoft Security Advisory 3042058，2015年5月12日）。

③ Switch to ECDSA hybrid certificates，https://phabricator.wikimedia.org/T86654#1405484（Brandon Black，2015年6月26日）。

④ Microsoft Achieves World Domination (in OCSP Stapling)，http://news.netcraft.com/archives/2013/07/19/microsoft-achieves-world-domination-in-ocsp-stapling.html（Netcraft博客，2013年7月19日）。

⑤ OCSP Stapling in IIS，http://unmitigatedrisk.com/?p=95（Ryan Hurst博客，2012年6月12日）。

- ❑ 缺少站点维度的配置

  IIS只有少数站点维度的SSL/TLS配置，也就是说，对于大部分配置（例如协议支持、密码套件顺序）来说，你需要找到一个符合所有站点的配置。通常这也不会成为一个问题，但确实会带来约束，尤其是在某些站点有特殊需求时（例如FIPS）。

密钥和证书的管理

IIS管理器的界面提供了基本的密钥和证书管理功能，虽然并不直观，但还是能用。以下我的说明和样例都是基于Windows Server 2012和IIS 8，但流程与更早（IIS 7和7.5）和更新（IIS 8.5）的版本应该基本相同。

> **注意**
>
> IIS用户界面中的用词并不十分准确。大部分标签和操作的名称都涉及证书，而你几乎总是会同时管理密钥和证书。为简单起见，这一节中我将使用IIS的术语。

1. 创建自定义IIS管理控制台

在开始实际的证书配置工作前，我建议你创建一个自定义Microsoft管理控制台（Microsoft management console，MMC）。

(1) 在“运行”菜单中，键入mmc来创建一个空的控制台。

(2) 从“文件”菜单中，选择“添加/删除管理单元”，会出现一个新窗口，左窗格是一个可用的管理单元列表。

(3) 添加“证书”管理单元，在第一个窗口中，选择“计算机账户”，在第二个窗口中，选择“本地计算机”。

(4) 添加“Internet信息服务”管理单元。

(5) 从“文件”菜单中，选择“保存”来保存当前控制台供之后使用，如果你把它保存到桌面，每次使用的时候只需要双击打开。

现在你就有了一个自定义控制台，可以用来管理网站证书，也可以用作IIS管理界面。

2. IIS证书管理

要使用IIS证书管理，打开IIS管理控制台并单击服务器名称，会出现一个有许多配置选项的窗格，其中一个配置选项是“服务器证书”（如图15-3所示）。

3. 创建自签名证书

创建自签名证书很简单：在右窗格中选择“创建自签名证书……”，并为其指定一个友好名称。你还可以选择证书存储的方式，可以放在个人证书库中或者放到网站证书库中，我现在还不清楚这两种方式有什么区别，我倾向于选择后者。

4. 导入证书

如果你已经有了一个证书，可以使用“导入”操作将证书导入到服务器，但唯一支持的格式是PKCS#12或PFX。如果你需要从使用了不同证书格式的Web服务器迁移，可以用OpenSSL对密

钥和证书格式做转换，相关内容在11.2.8节中有详细介绍。

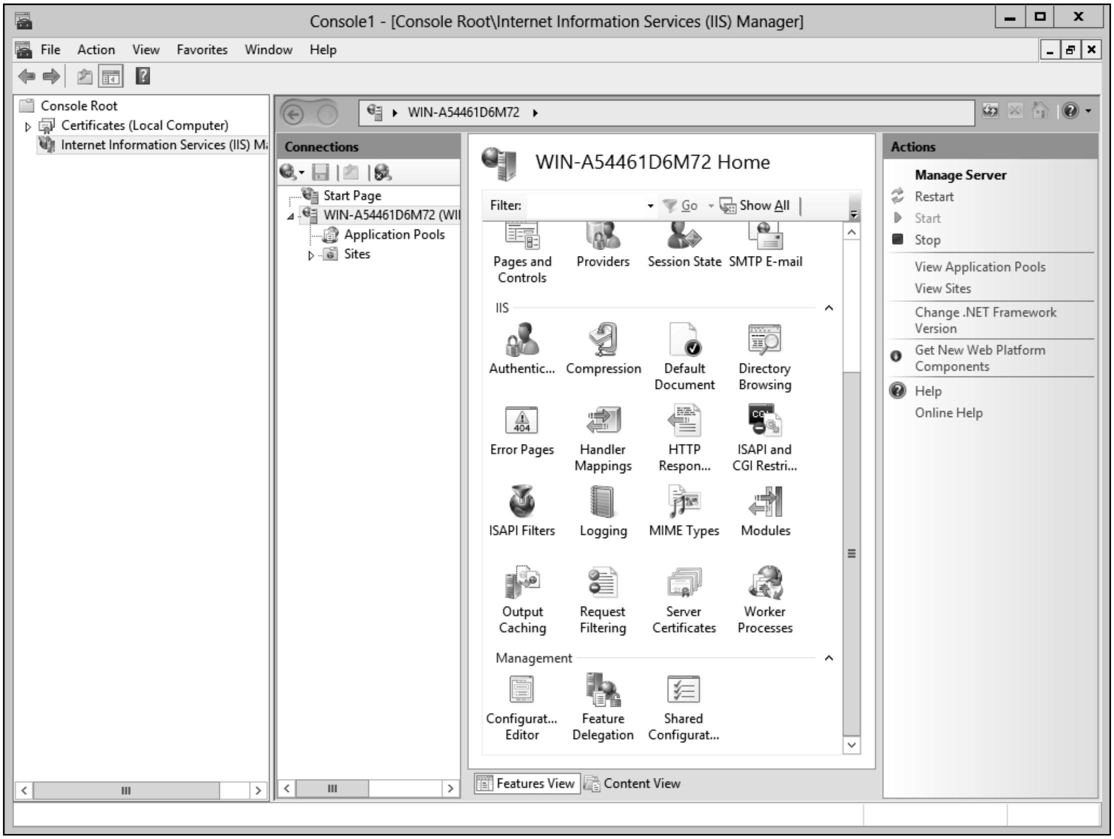


图15-3　IIS管理控制台中的服务器证书

> **警告**
>
> 当你导入证书时，最好禁用“允许导出此证书”选项，这样可以大大提高服务器密钥非法提取的难度。当然，如果禁用了这个选项，请务必在其他地方保留一份密钥和证书的备份。

5. 从公共CA申请证书

要从公共CA申请证书，首先需要创建一个证书签名申请（certificate signing request，CSR）。选择“创建证书申请”操作，会弹出一个包含以下三个步骤的操作向导。

(1) 在第一个页面中输入你的信息，确保关于你组织的信息是准确的，其中的“公用名”字段应该使用你网站的主域名。

(2) 在第二个页面中选择密钥的类型和强度。对于类型，默认值（Microsoft RSA SChannel

Cryptographic Provider）是目前唯一可行的选择，不需要更改。对于强度，选择2048位（我这里默认值是1024位，强度较弱且不安全）。

(3) 在第三个页面中选择CSR文件的存储位置。

现在你有了自己的CSR，接着就是要把它提交给所选择的CA。大多数情况下，你可以用文本编辑器打开CSR文件，将内容复制粘贴到CA网站的表单中。当CA对你请求的域名进行所有权确认后（对于DV域验证证书来说，通常过程是自动的且时间很短；对于EV扩展验证证书来说，过程会很长），就会签发你的证书。

> **警告**
> 在生成CSR文件时，你也同时在此计算机上创建了唯一的私钥。由于证书和私钥不匹配时是不能工作的，因此你应该确保对两者都做好安全的备份。最好的方法是在你需要部署证书的服务器上创建CSR和私钥，然后使用“导出”功能导出后进行备份。

6. 完成证书签名申请

处理完CSR请求后，CA通常会发给你几个证书，其中最主要的一个是站点证书，一般还需要一个或者多个中间证书，某些时候甚至需要根证书。如果CA把所有证书打包在一个文件里发送给你，你可以直接导入使用；如果CA将多个证书用不同的文件发送给你，你就需要将它们一个一个地导入，步骤如下所示。

(1) 如果CA根证书不在你的主可信证书库（称为可信根证书颁发机构，trusted root certification authority）中，导入根证书。

(2) 将所有中间证书导入到“中级证书颁发机构”证书库中。

现在，你可以使用“完成证书申请”操作来导入站点证书并匹配计算机上的对应私钥。如果已经正确配置了CA的根证书和中间证书，操作成功不会有任何警告，否则IIS会提示无法创建完整的可信证书链。

> **注意**
> 完成CSR的操作有时会提示失败，错误消息是“无法删除证书或访问被拒绝”。我也遇到了这个问题，但是最终发现实际过程已经成功完成，尽管有错误提示但证书可以正常使用。

7. 配置SSL站点

假设你已经有了证书，要对一个网站启用TLS，这时你需要对网站添加SSL绑定（SSL binding），从配置方面来讲有以下几个方面的内容。

- 协议：永远使用`https`
- IP地址和端口
- 域名
- “需要服务器名称指示”选项的正确配置（后面有更详细的说明）

- 需要的SSL证书

你可以有三种方式来配置一个安全的网站。

- 基于IP加端口组合的独立SSL站点

  传统意义上，每个安全网站需要部署唯一的IP加端口组合，但实际上为公共服务使用特殊端口并不可行，这个方案相当于为每个站点配置独立的IP地址。对于小规模托管服务来讲这个方案简单直接，但它需要你购买足够数量的IP地址。

- 证书共享

  尽管虚拟安全托管还不实用，但还是有办法在相同IP地址上为多个站点提供服务，只要你不介意让它们都使用相同的证书。你可以申请一个证书包含所有站点域名，或者申请一个泛域名证书用于支持所有匹配的子域名（或者同时使用这两种方式）。IIS 8全面支持这个特性，为一个站点配置SSL绑定时，选择好所需的IP地址和证书并输入正确的域名，对于不同网站不断重复这一过程，在此过程中SNI选项要保持禁用。

  IIS 8之前，在管理界面上只允许在一个IP地址加端口的组合上配置一个站点。但仍然有可能通过命令行直接修改配置来达到相同的效果，此外证书的友好名称中第一个字符必须使用星号（*）。[①]

- 虚拟安全托管

  在TLS协议最初阶段没有考虑对虚拟安全托管的支持，一些老的平台迄今仍然不支持这一特性，但其中的Windows XP仍然相当流行，因此我们必须将安全站点和IP地址绑定。IIS在版本8之后支持虚拟安全托管特性，你可以选中“需要服务器名称指示”选项来启用。然而，如果你还有用户在使用Windows XP上的IE浏览器，那么启用后这部分用户就无法成功连接到你的网站。如果你没有这样的老用户，那就可以安全地启用SNI特性。

8. 高级选项

本节中的说明仅适用于小规模部署，例如当你的服务器只提供少数站点服务时，需要处理复杂的网站架构就会变得相当困难。如果你的网站属于这一类，可以考虑使用以下这些高级选项。

- Web服务器集群的证书中心化

  从IIS 8.0开始，Web服务器集群管理变得更容易，因为它支持共享文件系统中的密钥和证书文件存储。[②]

- Active Directory和公共CA的集成

  近来公共CA开始提供基于自身公共架构的模拟私有CA产品，通过这种方法，许多工作（例如证书更新）变得简单和自动化。这个集成方案的优点是可以通过Active Directory策略来控制证书的签发，而证书最终会链接到公共CA。

① SSL Host Headers in IIS 7，https://www.sslshopper.com/article-ssl-host-headers-in-iis-7.html（SSL Shopper，2009年2月26日）。

② IS 8.0 Centralized SSL Certificate Support: SSL Scalability and Manageability，http://www.iis.net/learn/get-started/whats-new-in-iis-8/iis-80-centralized-ssl-certificate-support-ssl-scalability-and-manageability（IIS.Net，2012年2月29日）。

第16章

配置Nginx

16

Nginx是一个反向代理Web服务器，由于在系统资源使用上的高效率而广受欢迎。Nginx在当前稳定版本（1.6.x）上对TLS的支持良好，意味着你不会遇到TLS方面的任何问题（参见表16-1）。Nginx是一个相当年轻的项目，功能开发的节奏很快。如果你是一个高级用户，建议时刻关注开发分支上的功能改进。

表16-1　Nginx最新版本的TLS特性

功　　能	1.4.x	1.6.x	1.8.x
默认DH参数强度	默认1024位，较弱	默认1024位，较弱	默认1024位，较弱
DH/ECDH参数可配置	是	是	是
椭圆曲线（EC）算法支持	是	是	是
OCSP stapling	是	是	是
TLS分布式会话缓存	–	–	–
会话票证配置	–	是	是
禁用会话票证	–	是	是
后端证书验证	–	–	是

稳定版本的Nginx提供了部署一个独立运行、配置良好的TLS服务所需要的一切内容。默认的临时DH参数（1024位）的强度虽然比较弱，但这个问题可以通过配置解决。有一点需要特别注意的是，Nginx不会对反向代理的后端做证书验证。当后端服务器在本地时（例如在同一个网络中）这没有任何问题；但如果后端是一个公网服务器，这就是一个严重的安全缺陷。

本章旨在涵盖Nginx TLS配置中最重要和最有趣的方面，但它不是一个参考手册。如需更多信息，请参考官方文档。

16.1　以静态链接 OpenSSL 方式安装 Nginx

默认情况下，Nginx在安装时会自动检测系统的OpenSSL库并以动态链接方式使用。然而有时候你并不希望使用系统库，比如系统库的版本可能较低并且缺少一些重要特性。

实际上，你可以自己编译Nginx并且静态链接到一个兼容的OpenSSL版本。方法很简单：在配置Nginx的编译选项时，使用`--with-openssl`参数来指向OpenSSL的源代码：

```
$ ./configure \
--prefix=/opt/nginx \
--with-openssl=../openssl-1.0.1h \
--with-openssl-opt="enable-ec_nistp_64_gcc_128" \
--with-http_ssl_module
```

与其他一些程序不同（编译时指定的是OpenSSL静态库），Nginx需要指定源代码以便它可以自行配置和编译OpenSSL。在这种方式下，由于Nginx的隔离，你无法直接配置OpenSSL的编译选项。如果需要指定一个OpenSSL编译参数，可以通过Nginx的`--with-openssl-opt`参数来传入（例如在我的例子中，通过此参数启用了默认禁用的EC优化）。

> **警告**
> 使用源代码编译时要注意，与使用操作系统提供的包不同，你需要承担起软件更新的责任。如果发现安全问题，就需要及时重编译并更新部署。

16.2 启用 TLS

要启用TLS，你需要告诉Nginx在指定的端口上启用相应的协议。这可以通过在`listen`指令后加上`ssl`参数来实现：

```
server {
    listen 192.168.0.1:443 ssl;
    server_name www.example.com;
    ...
}
```

另一个建议设置的参数是`spdy`，用于启用SPDY协议[①]，要同时启用TLS和SPDY时，你可以这样配置：

```
server {
    listen 192.168.0.1:443 ssl spdy;
    server_name www.example.com;
    ...
}
```

16.3 配置 TLS 协议

当你启用TLS协议时，应该详细指定协议配置。不要相信默认设置，因为它们随时可能发生变化，时间一长，你就不清楚服务器实际在做什么了。Nginx协议配置主要有3个指令。第一个是`ssl_protocols`，用来指定启用的协议版本：

```
# 启用所有协议，禁用已废弃的
# 不安全的SSL 2和SSL 3
ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
```


① SPDY在默认编译选项中未启用，必须使用`--with-http_spdy_module`配置参数来启用。

第二个是ssl_prefer_server_ciphers，用来告诉Nginx在TLS握手时启用服务器算法优先，由服务器选择适配算法而不是让客户端来做：

```
# 让服务器选择要使用的算法套件
ssl_prefer_server_ciphers on;
```

最后一个是ssl_ciphers，用来指定使用的算法套件以及优先顺序，参数是OpenSSL套件名称，例如：

```
# 这个配置只启用了支持前向保密的算法，按照性能优先的顺序排列。
ssl_ciphers "ECDHE-ECDSA-AES128-GCM-SHA256 ECDHE-ECDSA-AES256-GCM-SHA384 ↩
ECDHE-ECDSA-AES128-SHA ECDHE-ECDSA-AES256-SHA ECDHE-ECDSA-AES128-SHA256 ↩
ECDHE-ECDSA-AES256-SHA384 ECDHE-RSA-AES128-GCM-SHA256 ECDHE-RSA-AES256-GCM-SHA384 ↩
ECDHE-RSA-AES128-SHA ECDHE-RSA-AES256-SHA ECDHE-RSA-AES128-SHA256 ↩
ECDHE-RSA-AES256-SHA384 DHE-RSA-AES128-GCM-SHA256 DHE-RSA-AES256-GCM-SHA384 ↩
DHE-RSA-AES128-SHA DHE-RSA-AES256-SHA DHE-RSA-AES128-SHA256 DHE-RSA-AES256-SHA256 ↩
EDH-RSA-DES-CBC3-SHA";
```

> **注意**
> 这个例子中的密码套件配置是安全的。根据实际偏好以及风险状况，你的配置可能会略有不同。在第8章中对TLS服务器配置有深入的讨论，在11.3.1节中的“推荐配置”部分中有OpenSSL的使用样例。

16.4　配置密钥和证书

配置安全服务的最后一步是指定所需的私钥和证书，你需要以下两个指令：

```
# 私钥
ssl_certificate_key server.key;

# 证书：服务器证书在最前面，后面是所有必要的中间证书，不需要根证书
ssl_certificate server.crt;
```

Nginx只有一个指令用于证书配置。如果你的服务器证书和中间证书是不同文件，需要把它们合并到一个文件来使用。请确保把服务器证书放在最前面，否则会得到一个配置错误；另外需要确保所有中间证书的顺序也正确，不然可能会导致一些很难发现的细小通信问题。[①]

> **注意**
> 尽管Nginx支持用密码来保护私钥，但唯一的方式是在程序启动时输入密码。因为需要交互，所以这一方案在实践中无法实施，只能配置一个没有密码的私钥，而这种部署不太理想。好消息是，1.7.3版本增加了一个新指令ssl_password_file，使用这个指令可以将私钥的密码加入配置文件中。

① ssl_certificate指令允许你将私钥和证书合并到同一个文件内，但是将私有和公有信息存储在同一个文件内是非常危险的，很容易导致私钥的泄露。

16.5 配置多密钥

目前Nginx还不支持对一个网站配置多个密钥；但如果你希望保持老客户端兼容性的同时又能获得最佳性能，多密钥功能就是必需的。如果可以部署多密钥，你可以对新的客户端使用更快的ECDSA，而对老的客户端使用兼容的RSA。Nginx从2013年11月开始已经完成了一部分这方面的工作，但是在官方Nginx代码库中还从未正式支持①。最近，维基百科使用了一个更新版本，其中就包含了一个自有的双证书（ECDSA+RSA）补丁。②

16.6 通配符证书和多站点证书

如果你有两个以上的站点共用相同的证书，就可以把它们部署在同一个IP地址上；不过虚拟安全托管的工作模式在现有的情况下还不可行。这无需特殊配置，只要把所有网站绑定到相同的IP地址上并确保它们使用了相同的证书即可。③

这种方式的工作原理是，TLS终止和HTTP域名处理是两个独立的处理步骤。当终止TLS时，Nginx会从指定IP地址的默认服务器（配置中的第一个服务器）中返回证书。当处理HTTP请求时，Nginx会检查Host请求头并转发到server_name配置与Host头相匹配的服务器。如果未匹配成功，则使用默认的服务器。

复用证书的最佳方式是把证书配置放在http段内，这样之后的服务器配置都可以直接继承：

```
# 为所有之后的服务器配置同一个证书和同一个密钥
ssl_certificate      server.crt;
ssl_certificate_key server.key;

# site1.example.com
server {
    listen      443 ssl;
    server_name site1.example.com;
    ...
}

# site2.example.com
server {
    listen      443 ssl;
    server_name site2.example.org;
    ...
}
```

① [PATCH] RSA+DSA+ECC bundles，http://mailman.nginx.org/pipermail/nginx-devel/2013-October/004376.html（Rob Stradling，2013年10月17日）。

② Switch to ECDSA hybrid certificates，https://phabricator.wikimedia.org/T86654（Wikimedia Phabricator #T86654，修复于2015年7月13日）。

③ 严格来说，证书部署的限制针对的是IP地址和端口的组合，例如可以在192.168.0.1:443上运行一个站点而在192.168.0.1:8443上运行另一个站点。在实际环境中网站只能运行于443端口，所以这一限制也就变成了不同的IP地址。

这种方式简化了维护并且在内存中只会保存一份证书和私钥的信息。

16.7　虚拟安全托管

与16.6节中讨论的不同，真正的虚拟安全托管指的是：多个不同的网站，使用不同的证书，共享一个IP地址。由于这一特性没有包含在SSL和TLS协议的早期版本中，迄今为止仍然有许多老客户端是不支持的。基于这一点来考虑，对于受众广泛的互联网站点，采用虚拟安全托管模式并不合适；反过来说，对于用户群都是现代浏览器的站点则完全可行。

Nginx支持虚拟安全托管模式并在需要时会自动启用。唯一的问题是，如果你按虚拟安全托管模式部署，当遇到不支持这一特性的用户时，你该怎么办？通常情况下，Nginx会对这种用户返回访问IP地址上默认站点的服务器证书，而这个默认证书并不一定能与用户的访问域名匹配，这时用户就会收到一个证书警告。如果用户忽略警告继续访问，他们仍然可以访问到正确的站点。[①]

16.8　默认站点返回错误消息

对于不正确的请求，返回网站内容从来都不是一个好主意！例如，你不会希望搜索引擎把网站收录到一个错误的域名下。更重要的是，缺少域名校验会带来网站迁移时的安全风险。因此，强烈建议只将默认站点用于返回错误消息而不提供实际服务。

这里有一个配置的例子供参考：

```
# 默认站点用来对不正确域名的请求返回错误消息
server {
    listen 443 ssl default_server;

    # 不需要配置server_name，因为不用匹配
    # 域名，所有未匹配的请求都会进入默认站点
    # server_name "";

    root /path/to/site/root;

    location / {
        return 404;
    }

    location /404.html {
        internal;
    }

    error_page 404 /404.html;
}
```

基于这个配置，当用户请求一个不存在的域名时就会收到404.html错误页。大多数情况下，

① 当然，这里假设请求的域名在HTTP层面上存在一个虚拟站点配置，如果没有，则会访问到默认站点。

用户需要点击通过一个证书警告，但服务器有时也可能对一个不存在的虚拟主机域名返回一个有效的证书，通常使用泛域名证书时就存在这个潜在问题。

在写作本书时，Nginx不支持严格的SNI检查，即发现用户不支持SNI时直接在HTTP通信时拒绝服务，无论域名是否正确。实际上，所有非SNI用户都可以点击证书警告来继续访问只支持SNI的站点。这种工作方式非常有用，可以让用户知道他们所遇到的问题。[①]

16.9 前向保密

Nginx在支持前向保密上没有任何问题。从2011年8月发布的1.1.0版本开始，Nginx就已经完全支持DHE和ECDHE的密钥交换。唯一要注意的是OpenSSL库是否支持EC算法，并不是所有版本都支持它，原因有下面两个。

- OpenSSL版本太老

  如果系统底层安装的OpenSSL不支持新功能（如EC加密），那么即使Nginx支持也无济于事。很多老系统普遍都安装了老版本的OpenSSL，即使一些新发布的操作系统也很可能带的是老版本的库，比如2013年11月发布的OS X Mavericks自带的是OpenSSL 0.9.8.y（0.9.x分支的最新版）。要支持EC加密算法，你需要安装1.0.1及以上版本的OpenSSL。

- OpenSSL的版本不支持EC

  很长时间以来，Red Hat的操作系统自带的OpenSSL不支持EC加密，因为他们的律师希望在EC加密成为可信任的专利后再使用。这让使用Fedora和Red Hat企业版（及其衍生版本）的用户难以部署前向保密。唯一的解决办法是重新编译OpenSSL和所有其他依赖包。

  2013年10月，情况有所改变，Fedora 18和之后的版本在发布时已经使用了支持EC加密的OpenSSL包[②]。2013年11月，Red Hat Linux 6.5企业版也开始支持EC加密[③]。

16.10 OCSP stapling

Nginx从1.4.x分支开始支持OCSP stapling功能。到目前为止，这一特性是被Nginx作为一个优化方式来实现的。就具体实现来说，Nginx不支持在启动时预加载OCSP响应；相反，它是在第一次连接建立时才发起自己的OCSP请求，其结果是，第一个连接永远都不会收到OCSP响应。此外，OCSP响应结果在Nginx的工作进程中是不共享的，也就是说只有当每一个工作进程都经历过上述流程后，Nginx服务才能全部就绪。

---

① 1.7.0引入了一个新的`$ssl_server_name`变量来表示SNI域名（如果请求中有这个字段），对于不支持SNI的客户端，值是空字符串。可以用这个变量在站点配置中检测请求是否正确并返回不同的错误页面，唯一的问题是你需要在每个站点配置中加入这个检查。

② Bug #319901: missing ec and ecparam commands in openssl package，https://bugzilla.redhat.com/show_bug.cgi?id=319901（Red Hat Bugzilla，修复于2013年10月22日）。

③ Red Hat Enterprise Linux 6.5 Release Notes，https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html-single/6.5_Release_Notes/（Red Hat，2013年11月21日）。

在实际环境中，因为从CA获取OCSP响应需要消耗一定的时间，所以可以认为服务器在启动后的一段时间内OCSP stapling功能不是完全可用的。服务器越繁忙，这个时间就越短。

这种延时在实践中并不会带来问题，因为OCSP stapling不是一个必要环节；浏览器如果能收到OCSP响应时就会直接处理，如果没有收到就会自己从CA拉取。如果你觉得对每次连接都返回OCSP响应是必要的，可以考虑在Nginx中手动设置，本节的后面会有这方面的讨论。

> **警告**
> 由于内部bug[①]，Nginx有可能会发送过期的OCSP响应。主要问题在于Nginx内部的OCSP响应更新进程是固定每一小时执行一次，与实际响应中CA设置的过期时间无关。如果服务器收到的OCSP响应的过期时间在一小时之内，就会有那么一段时间Nginx会返回过期的OCSP响应。此bug在1.9.x分支中已修复。

16.10.1　配置 OCSP stapling

要使用OCSP stapling，你只需要告诉Nginx启用这一特性：

```
# 启用OCSP stapling
ssl_stapling on;

# 配置一个DNS服务器用于解析域名的IP地址
resolver 127.0.0.1;
```

> **注意**
> OCSP请求是通过HTTP提交的，这要求你的Web服务器能够向互联网上的OCSP服务商发起出站请求。如果你有一个出站流量的防火墙，要确保将这种类型的请求配置为允许出站。

建议你同时启用OCSP响应验证，默认情况下此项是禁用的，因此需要多一点配置工作，但能确保提供给用户的响应是唯一合法有效的：

```
# 启用OCSP响应验证
ssl_stapling_verify on;

# OCSP响应验证需要完整的证书链，因此需要配置所有的中间证书以及
# 根证书，一般情况下根证书在服务器证书中是不包含的
ssl_trusted_certificate trusted-for-ocsp.pem;
```

很容易发现Nginx的OCSP stapling配置中缺少缓存和超时相关的指令。缓存功能无需任何配置，OCSP响应不支持多进程之间共享，但每个工作进程都有自己的内存缓存并会按需增长。对于超时时间，Nginx全都是硬编码：有效响应的缓存时间是1小时，错误响应的缓存时间是5分钟，

① ocsp stapling may send expired response，https://trac.nginx.org/nginx/ticket/425（Nginx Bug #425，检索于2014年7月10日）。

网络超时时间是60秒。[①]

16.10.2 自定义 OCSP 响应

通常情况下，会根据不同证书将OCSP请求提交给相应的OCSP服务商。然而在以下两种情况下，你可能希望自己配置OCSP服务商。

- 在一个严重封闭的环境中，可能不允许直接从Web服务器发起外连请求。这种情况下，需要为OCSP请求配置一个转发代理来支持OCSP stapling。
- 某些证书可能没有内置OCSP服务商的信息，尽管签发证书的CA实际有相应的服务。这种情况下，可以手动设置一个OCSP响应URI。

你可以使用ssl_staping_responder指令，在全局或者站点的配置中设置OCSP服务商信息：

```
# 为OCSP请求设置一个转发代理
ssl_stapling_responder http://ocsp.example.com;
```

16.10.3 手动配置 OCSP 响应

如果你想为所有安全连接提供可靠一致的OCSP响应，需要人工处理OCSP响应的获取和更新，Nginx只需要负责给用户返回已配置的响应。

在Nginx配置中，使用ssl_stapling_file指令来指定一个以DER格式存储的OCSP响应文件：

```
# 设定Nginx从文件中获取OCSP响应而无需从服务商
# 拉取，由人工负责这个文件内容的更新
ssl_stapling_file ocsp-response_www.example.com.der;
```

获取OCSP响应最简单的方法是使用OpenSSL命令行工具。在开始前，你需要准备好服务器的证书以及签发CA的证书。颁发者的证书应该已经包含在下发给你的中间证书中了；但也可能你的颁发者直接使用了根证书签发（实际情况中非常罕见），这种情况你需要找你的CA直接要一份。

你的下一个任务是找到OCSP响应程序的地址，可以通过检查服务器证书中的颁发机构信息访问（authority information access，AIA）扩展获取。例如：

```
$ openssl x509 -in server.crt -noout -ocsp_uri
http://rapidssl-ocsp.geotrust.com
```

接下来通过这个URL和准备好的两个证书，你可以提交一个OCSP请求：

```
$ openssl ocsp -issuer issuer.crt -cert server.crt -url http://rapidssl-ocsp.geotrust.com -noverify
    -respout ocsp-response_www.example.com.der
server.crt: good
    This Update: Jan 10 08:15:33 2014 GMT
    Next Update: Jan 17 08:15:33 2014 GMT
```

16

① OCSP stapling patches，http://mailman.nginx.org/pipermail/nginx-devel/2012-September/002682.html（Maxim Dounin，2012年9月5日）。

> **注意**
> 通常人工获取OCSP响应不会有什么问题，但在某些边缘情况下有可能会产生混乱，在12.9节中有这方面的详细信息。

这样你就有了一个包含合法OCSP响应的指定文件。尽管这种方法从理论上讲行得通，但在实际的生产环境部署中还需要更多考虑：你需要处理错误的场景，定期执行以更新OCSP响应数据，当任意文件变化时重新加载Nginx。

16.11　配置临时 DH 密钥交换

当涉及Diffie-Hellman（DH）密钥交换的强度时，Nginx默认提供1024位的安全性。这不够强，在国家级攻击面前可能也不够安全。要了解有关DH安全状态的更多信息，请参阅6.5节。

幸运的是，可以很容易地配置DH密钥交换的强度。要将DH密钥交换提高到2048位安全性，可以使用ssl_dhparam指令指定一个更高加密强度的参数文件：

```
# 使用一个更高位数的参数文件，而不是用默认的1024位
ssl_dhparam dh-2048.pem;
```

使用下面的OpenSSL命令生成2048位的参数文件：

```
$ openssl dhparam -out dh-2048.pem 2048
```

提高DH参数强度可能会影响到一些客户端的互通性。例如，Java 6和Java 7不支持1024位以上的DH参数，因此有可能导致连接直接失败。实际上，Java 7客户端在确保ECDHE套件优先时是可以成功连接的，对于Java 6客户端就没有什么办法了。如果你出于兼容性原因决定保留1024位DH参数，应当在每个服务器上生成唯一的DH参数，这可以确保这些参数不会轻易地被Logjam攻击所利用。

> **提示**
> 从安全的角度来看，你应当选择与服务器私钥匹配的DH参数强度。在实践中，大部分网站使用的是2048位私钥，因此理论上对所有人来说使用2048位DH密钥交换就已经足够了。不推荐使用更强的DH参数，因为那会显著降低TLS握手性能。

16.12　配置临时 ECDH 密钥交换

临时ECDHE密钥交换的默认强度是256 EC位，使用secp 256r1曲线模型（OpenSSL把它称为prime256v1），加密强度相当于3072位RSA密钥，足够安全，所以你不需要做任何改变。如果你确实需要调整，可以使用ssl_ech_curve指令：

```
# 使用更高192位安全强度的算法，相当于7680位RSA密钥
ssl_ecdh_curve secp384r1;
```

到目前为止，曲线模型的选择范围很小。尽管OpenSSL和其他平台可能支持多个曲线模型（就

OpenSSL而言，你可以用openssl的-list_curves参数获得完整列表），但只有secp256r1和secp384r1这两种是被浏览器广泛支持的，其中secp256r1在OpenSSL中有优化，所以运行得更快，secp384r1则没有专门的优化。

16.13　TLS 会话管理

Nginx提供独立服务器上TLS会话恢复的良好支持，包括服务器端会话缓存和会话票证两种方式。不过在服务器集群环境中，Nginx支持分布式会话票证，不支持分布式会话缓存。

16.13.1　独立会话缓存

对于独立服务器部署（通常有多个工作进程），你可以配置一个共享内存缓存以便TLS会话信息可以在工作进程中共享。Nginx默认没有启用TLS会话缓存，这会导致性能降低。

要启用会话缓存，你需要分配一块内存并且设定会话缓存的最长存活时间：

```
#  分配1MB的共享内存缓存
ssl_session_cache shared:ssl_session_cache:1M;

# 设置会话缓存过期时间为24小时
ssl_session_timeout 1440m;
```

尽管很难推荐一个适用于每一个人的缺省配置，但例子中的参考配置应该能够满足大部分网站的需求。使用1 MB的内存可以缓存大约4000个会话。

默认的会话缓存过期时间只有5分钟，这个时间太短，我推荐使用24小时。如果没有特殊理由，一般情况下不需要限制会话缓存过期时间，因为你总是希望缓存尽量多的TLS会话。在内存空间不足时，最老的会话会被自动清理用于新会话的缓存。不推荐使用24小时以上的时间，因为这可能会导致安全隐患。

Nginx为缓存配置提供了极大的灵活性。例如，你可以让同一个站点使用多层缓存，也可以让多个站点共用一个缓存。从安全角度考虑，每个站点应当使用独立的会话缓存，会话缓存共享应该仅限于同一个业务同一个证书的多个子站点中。关于这方面潜在风险的完整讨论可以参考6.8节。

16.13.2　独立会话票证

默认情况下会话票证是由OpenSSL处理的，Nginx配置不是必要的。对于独立服务器而言，这种方案确实“还行”，但仍然有以下这些方面是你应该关注的。

- 会话票证使用的是128位AES加密，Web服务器在启动初始化时会自动生成一个一次性密钥，根据服务器配置的不同，可能会有多个票证密钥同时使用。
- 票证密钥长度是固定的128位，对大部分场景都足够用了。
- 每次服务器重启时都会生成新的票证密钥，这会导致重启后所有进入的连接都需要重新进行TLS会话协商，因此会带来性能损失，但一般并不会有很大影响。

- 如果服务器长时间运行并且没有重启，运行期间所有的会话票证都会一直使用相同的AES密钥来加密。不推荐用这种方式运行服务器[①]，应当保证服务器定期重启，比如每天一次。

对于会话票证的安全性，最佳方案是为每个站点分配一个独立的票证密钥。

16.13.3　分布式会话缓存

到目前为止，Nginx尚不支持分布式会话缓存。2011年曾经发布过一个Nginx 0.8.x版本的补丁，增加了这个功能[②]，但是在之后的版本中被废弃。据一位Nginx开发人员介绍[③]，这个补丁以阻塞模式工作，与Nginx的事件驱动模型架构有冲突；也就是说当通过网络查询缓存时，所有Nginx工作进程都会被挂起（影响所有进入的请求），导致严重的性能损失。

由于Nginx不支持分布式会话缓存，这会影响到你的集群设计。集群部署上不可以自由地将新连接分发到任意节点；反之，你需要设计一种连接保持机制，让一个用户的访问保持分发到相同节点上[④]，这样这个节点上的独立内存会话缓存就能发挥作用了。

16.13.4　分布式会话票证

从1.5.7版本开始，Nginx就支持会话票证密钥的手动配置了。使用这个特性，你可以自己设计票证密钥的独立轮转机制或者在整个Web服务器集群中共享同一个票证密钥。

相关的指令是ssl_session_ticket_key，可以用它来设置票证密钥：

```
# 显式配置会话票证密钥
ssl_session_ticket_key ticket.key;
```

一个会话票证密钥由48字节的随机加密数据组成，分为3个16字节（128位）的片段，分别用于密钥名称、HMAC密码和AES密钥。这与OpenSSL内部的使用格式并不相同，因此你可能无法与其他Web服务器共享这个密钥。[⑤]

使用以下OpenSSL命令来生成一个新的密钥文件：

```
$ openssl rand -out ticket.key 48
```

在实践中，你应该至少配置两个密钥：用于生成新票证的主密钥，以及只用于解密的前一个

---

① 使用会话票证时，该AES密钥用于对所有会话数据（包括主密钥，它可以用于解密所有通信）进行加密，之后该信息会通过网络传输到客户端。这也使这个AES密钥成为了一个新的攻击点，当AES密钥被破解时，向前保密也会失效。

② SSL Session Caching (in nginx)，http://www.hezmatt.org/~mpalmer/blog/2011/06/28/ssl-session-caching-in-nginx.html（Matt Palmer，2011年6月28日）。

③ Re: Distributed SSL session cache，http://mailman.nginx.org/pipermail/nginx-devel/2013-September/004216.html（Maxim Dounin，2013年9月16日）。

④ 通常这是负载均衡设备的功能，通过记录原始请求的会话ID信息，随后将相同会话ID的请求转发到同一个后端服务器上。

⑤ NGINX SSL Session Ticket Key，http://mailman.nginx.org/pipermail/nginx/2014-February/042474.html（ZNV，2014年2月25日）。

密钥。

```
# 设置主密钥，用于新会话的加解密
ssl_session_ticket_key current-ticket.key;

# 保留前一个密钥用于老会话的解密
ssl_session_ticket_key previous-ticket.key;
```

使用两个密钥轮转的方式，在密钥更新时服务器就不会丢弃更新前建立的会话。

在集群中实施会话票证密钥的轮转是不可靠的，因为这要求新的密钥在同一个时刻被部署到所有节点上。如果有一个节点在其他节点前先启用了密钥，其他节点就有可能无法解密数据，从而导致SSL握手重建。不过这种情况应该不是问题，除非你会频繁地更新密钥。此外，许多集群在设计上会把同一个用户保持分发到相同节点，上面说的问题也就不存在了。

如果你坚持要完美地实现集群的会话票证密钥轮转，并且不介意操作两次集群配置，可以按以下步骤操作。

(1) 生成一个新的会话票证密钥。

(2) 将新的密钥替换掉只用于解密的老密钥，加载集群配置，这样所有节点都可以解密新密钥的数据。

(3) 重新配置集群，将两个密钥交换位置，新密钥作为加解密的主密钥，之前的主密钥作为只解密的老密钥。因为所有节点在上一次配置中已经加载了新密钥，所以会话可以保持正常工作，不会有任何时间差带来的问题。

16.13.5 禁用会话票证

从1.5.9版本开始，Nginx允许禁用会话票证。当你使用的是服务器集群且不希望部署共享票证密钥时，这个选项可以提供帮助：

```
# 禁用会话票证
ssl_session_tickets off;
```

如果你用的是更早版本的Nginx，可以从开发列表的归档中获取相关补丁。[①]

16.14 客户端身份验证

想在配置中启用客户端身份验证，Nginx需要提供一个完整证书链的所有CA证书，以及一份相关的证书吊销列表。下面是一个完整的示例：

```
# 要求客户端身份验证
ssl_verify_client on;

# 指定客户端证书到根证书的最大证书路径长度
ssl_verify_depth 2;
```


① [PATCH] SSL: ssl_session_tickets directive，http://mailman.nginx.org/pipermail/nginx-devel/2014-January/004760.html（Dirkjan Bussink，2014年1月10日）。

```
# 指定允许签发客户端证书的CA证书，证书名称
# 将被发送给用户用于客户端证书选取
ssl_client_certificate sub-ca.crt;

# 完整证书链中需要包含的其他CA证书
ssl_trusted_certificate root-ca.crt;

# 证书吊销列表，文件有更新时Nginx需要重新加载
ssl_crl revoked-certificates.crl
```

有了这些配置，Nginx将只接受包含有效客户端证书的请求。如果请求未包含证书或者证书校验失败，Nginx会返回一个400错误响应。

除了启用强制客户端身份验证之外，ssl_verify_client还有另外两个设置在某些情况下有用：

- optional

  启用客户端证书但身份验证失败时不终止TLS握手，验证结果会被保存在$ssl_client_verify变量中：NONE表示没有证书，FAILED表示证书未通过验证，SUCCESS表示证书有效。如果想让客户端身份验证失败时返回自定义响应，这个选项是非常有用的。

- optional_no_ca

  启用客户端证书但在TLS握手时不进行验证，证书验证工作交由外部服务来负责（证书保存在$ssl_client_cert变量中）。

> **注意**
>
> 使用可选的客户端身份验证方式是有问题的，因为有些浏览器在这种情况下不会提示用户选择客户端证书，还有一些浏览器在无法提供客户端证书时将不能访问网站。当你确实考虑要部署可选的客户端身份验证方式时，请对环境中可能遇到的所有浏览器都做好测试。

16.15　缓解协议问题

Nginx的用户无需担心遇到SSL和TLS协议上的问题，所有问题在发现后都得到了快速解决，有一次甚至在问题被公开前就已修复。

16.15.1　不安全的重新协商

不安全的重新协商是2009年11月发现的一个协议上的缺陷，并在2010年基本得到了解决。在问题发现的一周内，Nginx就发布了0.8.23版本，修复了这个问题。自此以后客户端发起的重新协商就不再有效了。

此外，Nginx也不支持服务器发起的重新协商。服务器重新协商通常用于当同一个站点运行于多个安全环境时，例如，你可能允许任何人访问网站的首页但是在访问下一层的子页面时需要客户端身份验证。Nginx支持客户端身份验证，但是只能在服务器级别配置（不支持子目录的配置），也就意味着服务器重新协商是无意义的。理论上，Nginx在编译时选择好OpenSSL版本是可

以支持安全重新协商功能的，只是现在的实现方式是直接拒绝重新协商请求。

16.15.2 BEAST

严格来说，TLS 1.0和更早版本中的IV可预测漏洞会同时影响客户端和服务器端，但实际上只有浏览器会受到攻击（所谓的BEAST攻击），因为这种攻击需要攻击者可以控制受害者发送的数据（并且之后就被加密了）。基于上述原因，服务器代码在这个漏洞修复上起不了什么作用。

16.15.3 CRIME

2012年的CRIME攻击利用的是TLS协议启用压缩时的信息泄露[①]，至今这个问题还没有解决（在TLS压缩启用时），变通的方法是完全禁用压缩。考虑到性能问题，Nginx开发者从2011年就已经开始禁用压缩，但最初的修改（1.0.9和1.1.6版本）只支持OpenSSL 1.0.0及以上版本。从2012年的1.2.2和1.3.2版本开始，Nginx对所有OpenSSL版本都已经支持禁用压缩。[②]

16.16 部署 HTTP 严格传输安全

由于HTTP严格传输安全（HTTP strict transport security，HSTS）是通过响应头来激活的，因此使用起来非常容易，但也有不少使用上的陷阱。因此我建议你在作出决定前，先仔细阅读10.1节。

一旦站点启用了HSTS，用户的后续访问就会直接进入443端口，然而你还需要确保那些访问到80端口的用户能被重定向到正确的地址。为了支持这个重定向，而且由于在明文响应中HSTS响应头是不被允许的[③]，你需要配置两个不同的服务器，例如：

```
server {
    listen 192.168.0.1:80;
    server_name www.example.com;

    return 301 https://www.example.com$request_uri;
    ...
}

server {
    listen 192.168.0.1:443 ssl;
    server_name www.example.com;

    add_header Strict-Transport-Security
        "max-age=31536000; includeSubDomains; preload";

    ...
}
```

① TLS不是唯一受影响的协议，信息是否泄露取决于压缩的具体实现，在其他网络层也可能存在。例如，在HTTP响应中使用的gzip算法也有漏洞。

② crime tls attack，http://mailman.nginx.org/pipermail/nginx/2012-September/035600.html（Igor Sysoev，2012年9月26日）

③ 如果明文响应中允许设置HSTS头，中间人攻击者就可以通过在普通站点中注入HSTS信息来执行DoS攻击。

关于Nginx的add_header指令的作用，有两个方面需要关注。第一，HTTP头只会被添加到非错误响应中（即2xx和3xx），对于HSTS这不是什么问题，因为大部分的正常响应都在这个范围内。第二，这个指令的继承方式有时很让人费解：如果一个子配置块中设置了add_header指令，那么在上层配置块中的add_header指令是不会被继承的。换句话说，如果你需要在子配置块中添加add_header指令，还需要从上层配置块中复制所有的add_header指令。

16.17　TLS 缓冲区调优

从1.5.9版本开始，Nginx允许使用ssl_buffer_size指令自定义TLS缓冲区的大小。默认值是16 KB，但是这个值不一定是最优化的，尤其是你希望首字节数据被尽早发送时，有报告显示使用1400字节的配置可以显著减少延迟。①

```
# 减少TLS缓冲区大小，可以显著减少首字节时间
ssl_buffer_size 1400;
```

需要注意的是，减少TLS缓冲区大小有可能会降低连接的吞吐量，特别是当你需要发送大量的数据时。②

16.18　日志记录

默认的Web服务器日志只记录错误信息以及访问的地址，并没有多少TLS使用上的记录，然而通常有以下两方面的原因需要你关注服务器上的TLS使用情况。

- 性能

  TLS会话恢复的配置错误会导致严重的性能损失，因此你需要特别关注会话恢复率，在日志中做好记录可以帮助你了解会话恢复的使用情况以及优化会话缓存配置。

  从1.5.0版本开始，Nginx支持$ssl_session_reused变量，可以帮助你直接跟踪会话恢复。如果你使用的是之前的版本，就只能通过日志后期处理的方式，统计日志中相同会话ID出现的次数。根据会话恢复率，你可以真实地了解TLS会话缓存的工作性能。

- 协议和密码套件的使用

  掌握你的用户群所使用的协议版本和密码套件是非常重要的，原因有二：(1) 你需要确保配置中的设想都是正确的；(2) 你需要了解一些旧特性是否仍然需要保留，例如SSL 2的支持就被保留了很长时间，因为人们担心关闭后有影响。时至今日，我们在SSL 3协议以及RC4和3DES加密算法的选择上又面临着类似的问题。

最好是使用一个独立的日志文件来记录TLS连接信息。在Nginx中这需要使用两条指令，分

① Optimizing NGINX TLS Time To First Byte (TTTFB)，https://www.igvita.com/2013/12/16/optimizing-nginx-tls-time-to-first-byte/（Ilya Grigorik，2013年12月16日）。

② Optimizing NGINX TLS Time To First Byte (TTTFB)，http://mailman.nginx.org/pipermail/nginx/2013-December/041502.html（Nginx开发人员邮件列表讨论，2013年12月16日）。

别用于定义一个新的日志格式以及生成新的日志文件：

```
# 设置TLS日志格式，变量$ssl_session_reused在1.5.10及以后版本中支持
log_format ssl "$time_local $server_name $remote_addr $connection $connection_requests $ssl_protocol ↩
$ssl_cipher $ssl_session_id $ssl_session_reused";

#记录TLS连接信息
access_log /path/to/ssl.log ssl;
```

> **警告**
> 由于Nginx中的一个bug，1.4.5和1.5.9之前版本中的$ssl_session_id变量没有包含正确的TLS会话ID。如果需要记录会话ID，你需要升级到更新的版本。

这种类型的日志方式将会为每一个成功处理的HTTP会话生成一条记录，某种意义上说这是一种浪费，因为所有的TLS参数其实在首次建立连接时就已经确定了（Nginx不支持会话重新协商，参数也就不会被改变）。另一方面，连接复用是最高效的工作方式，因此跟踪连接复用情况是很重要的，所以我在日志格式的参数中加入了$connection和$connection_requests两个变量。

> **注意**
> 目前还没有办法记录那种没有实际HTTP请求但TLS握手成功的信息，同样也没有办法记录TLS握手失败的信息。

第 17 章

总　　结

恭喜你，一路完成了本书的学习！希望你在阅读时能像我写作时那样，收获一样多的乐趣。我们在书中讨论了这么多的TLS安全问题，那么我们到底面临着怎样的现状呢？TLS安全吗？还是说它有着无法修复的缺陷而注定要消亡？

与其他许多问题类似，答案完全取决于你对TLS的期望。如果与一些想象中的理想产品相比较，很容易指出TLS中的各种问题；TLS也确实存在很多问题，整个社区长期以来一直都在努力地修复完善。然而，安全协议的成功不能单纯地从技术性和安全性方面来衡量，更加重要的是在现实生活中的成功实践和实际效果。因此，尽管TLS并不完美，但每天仍有数十亿人使用它，这已经是一个巨大的成功。如果一定要在TLS生态系统中选出一个最大的问题，那就是我们还没有充分地使用加密，使用的时候也没有认真思考我们是否真的安全（想一想证书警告）。TLS的缺陷反而不是什么大问题。

我们一直在讨论TLS的安全性，这其实正是因为TLS非常成功，不然它早就被其他更好的产品取代了。不过，即使我们有机会使用其他产品来代替TLS，在经过多年的使用后，我们一样会碰到与TLS现状相同的情况。我清楚地意识到在全世界范围内不可能达到所谓完美的安全性。这个多样性的世界正在加强安全性方面缓慢前进，同时尽量避免对现状造成重大的破坏。你知道吗？这其实没什么大不了。这就是加入全球计算机网络的代价。

好消息是TLS正处在一个不断改善的良好阶段。多年前的某个时候，我们开始将更多注意力放在安全性上，尤其是加密环节。这一过程在2013年开始加速，因为随着用户使用越来越广泛，我们也不断地遭遇到大量安全问题。TLS工作小组正在忙于开发下一个协议版本。这一版本不会有（也不需要有）根本上的不同，但却会把我们的安全水平提高到一个更高的阶段。我会把这些新的内容写在这本书的未来版本中！

Bulletproof SSL and TLS

Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications

HTTPS权威指南

在服务器和Web应用上部署SSL/TLS和PKI

本书主要内容：

- 介绍密码学和最新的TLS协议版本
- 讨论各个层面上的弱点，涵盖实施问题、HTTP和浏览器问题以及协议漏洞
- 分析最新的攻击，如BEAST、CRIME、BREACH、Lucky 13、RC4、三次握手和心脏出血
- 提供全面的部署建议，包括严格传输安全、内容安全策略和钉扎等高级技术
- 使用OpenSSL生成密钥和证书，创建私有证书颁发机构
- 使用OpenSSL检查服务器漏洞
- 给出使用Apache httpd、IIS、Java、Nginx、Microsoft Windows和Tomcat进行安全服务器配置的实际建议

沃通电子认证服务有限公司（WoSign）审读

Feisty Duck LONDON

图灵社区：iTuring.cn

热线：(010)51095186转600

分类建议 计算机/Web安全

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-43272-8

ISBN 978-7-115-43272-8

定价：99.00元